General Disclaimer

One or more of the Following Statements may affect this Document

- This document has been reproduced from the best copy furnished by the organizational source. It is being released in the interest of making available as much information as possible.

- This document may contain data, which exceeds the sheet parameters. It was furnished in this condition by the organizational source and is the best copy available.

- This document may contain tone-on-tone or color graphs, charts and/or pictures, which have been reproduced in black and white.

- This document is paginated as submitted by the original source.

- Portions of this document are not fully legible due to the historical nature of some of the material. However, it is the best reproduction available from the original submission.

NASA CR. 143992

FORTRAN AUTOMATED CODE EVALUATION SYSTEM (FACES)

SYSTEM DOCUMENTATION

Version 2, Mod 0

IBM Host - ModComp FORTRAN

September, 1975

(NASA-CR-143992) FORTRAN AUTOMATED CODE EVALUATION SYSTEM (FACES) SYSTEM DOCUMENTATION, VERSION 2, MOD 0 (Brown and Ramamoorthy, Inc., Berkeley, Calif.) 594 p HC $13.25 CSCL 09B G3/61

N75-33746

Unclas 42348

Contract : # NAS8-30928

BROWNE & RAMAMOORTHY, INC.

2550 Telegraph Avenue, Suite 404, Berkeley, Ca., 94704

(415) 848-0261

Table of Contents

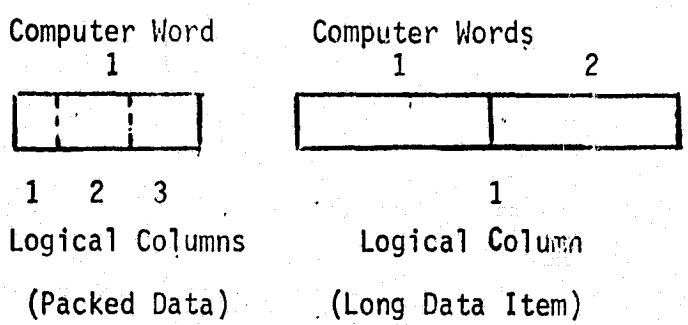
Definitions

Computer Word - The word size of an integer word on the host machine.

Host Machine - Computer system on which the FACES system is run.

Local Tables - Analysis tables constructed by FACES to describe the activities of a module. The Local Tables are the Symbol Table (with overflow), Use Table, Node Table, Successor Table, and Predecessor Table.

Logical Column - Logical columns indicate the association of table entries which require different physical word storage. If multiple data items are packed in a single word, several logical columns are represented in a word. If a data entry is longer than one computer word, several words may be used for one logical column.



Module - FORTRAN subprogram presented for analysis. May be a main program, subroutine, function, or BLOCK DATA.

Operand -	A FORTRAN variable or constant used in a module.
Target FORTRAN -	FORTRAN dialect for which FACES has been adapted.
V( ) Form -	Element of a FORTRAN statement indicating either an array or function reference.

I. FACES OBJECTIVES

The FACES system processes FORTRAN based software systems to surface potential problems before they become execution malfunctions. Analysis services complement the diagnostic capabilities of compilers, loaders, and execution monitors rather than duplicate these functions. FACES emphasizes frequent sources of FORTRAN problems which require inordinate manual effort to identify.

The principle value of FACES is extracting small sections of unusual code from the bulk of normal sequences. Code structures likely to cause immediate or future problems are brought to the user's attention. These messages stimulate timely corrective action of solid errors and promote identification of "tricky" code. Corrective action may require recoding or simply extending software documentation to explain the unusual technique.

Effort is directed toward identifying errors which would produce legal FORTRAN programs subject to malfunction. These problems are extremely difficult to identify and correct.

1. Legal interfaces among modules which appear suspicious.
2. Code subject to misinterpretation during maintenance and modification.
3. Algorithm independent illogical operations.
4. Keypunch errors likely to escape compilation diagnosis.
5. Hazardous calling sequences among modules.

Subsystem Objectives. FACES is composed of 4 components: Control Driver, FORTRAN Front End, Automatic Interrogation Routine, and Report Generator. Each component has subordinate objectives in accomplishing code analysis.

1. Control Driver. The Control Driver performs file positioning and housekeeping functions to coordinate resident data between runs and coordinate subsystem results. The control routine is also responsible for interpreting command control cards.
2. FORTRAN Front End (FFE). The FFE processes new FORTRAN source code providing analysis tables and a catalogued file of module source text. The FFE provides FORTRAN interpretation to the source code instructions provided. Error detection is limited to code features or system constraints which impact analysis scope.
3. Automatic Interrogation Routine (AIR). AIR inspects the analysis tables for specific constructions under user selection control. Where constructions of interest are found, message flags are recorded for use in generating reports.
4. Report Generator. The report generator constructs user displays of the analysis results.

II. OVERVIEW OF SYSTEM OPERATION

FACES incorporates aspects of compilation, file management, and data base interrogation to provide an analysis system suitable for investigating software systems. Since some code normally becomes available before a runnable configuration is achieved, FACES adapts its operation to analyze whatever code is currently available. As new modules are added, the new source code is included in the analysis. For ease of operation, old modules are maintained between runs, allowing incremental addition to the software data base.

The system can be initialized to an empty state for the first run, creating an empty directory of modules, empty source code catalogue, and empty analysis table file. The first set of FORTRAN modules is analyzed, creating directory entries, analysis tables for each module, and module source text added on a Source Code Catalogue. After the initial system has been created, new modules can be added or existing modules replaced with new versions. File status is automatically updated on each run.

Analysis is requested by the user either when adding source or in a stand-alone run. The Automated Interrogation Routine performs the selected analysis using information created by FFE on the analysis table file. In addition to servicing user requests, AIR also performs "system queries" which construct global data for the system extracted from the current module set local descriptions.

During analysis, AIR reports findings of investigations on a Flag file. Flag file entries identify the source code lines participating

in the result and provide support data to be included in the diagnostic message.

The Report Generator collects the information generated on the Flag file and produces user reports of the analysis results. These reports consist of messages generated by the FFE and AIR associated with the source code card images obtained from the Source Code Catalogue.

To consolidate report information, a sort of the Flag file is required before actually producing the report. The sort action permits redundant messages to be suppressed while printing the report, and association of several messages with the same source line.

III. DESIGN REQUIREMENTS

The following is a list of design requirements which influenced the selection of techniques for construction of FACES.

1. Historical Influence. FACES Version 2 is adapted from methodologies and techniques developed from FACES Version 1. To minimize development cost, a complete redesign of the system was excluded. Enhancements were limited to extensions and refinements of established processing methods.
2. Transportability. The FACES system is constructed for transportation to different operating environments.
3. Maximum Acceptance of FORTRAN Dialects. FORTRAN analyzed is primarily ANSI Standard text with common compatable extensions included in the analysis. Where conflicting forms of statement exist on several machines, statements are excluded from the analysis.
4. System Extension. The FACES system is designed for extension of capabilities. Coding techniques allow for the possible addition of capabilities while maintaining core processing intact.
5. Operational Simplicity. Coding techniques emphasize simplicity in processing and ease of understanding rather than optimal execution speed. Speed improvement is anticipated through improvement of time dominate routines found after installation.
6. Processing Continuity. The user should receive some benefit from each run. Where limitations on processing occur, the

procedure is to truncate processing and continue. Only operational impasses result in abort termination.

7. Familiarity in Result Reports. Since the user's primary familiarity with the subject software system is a compiler listing, the source code listing is the primary vehicle for reporting results. Extracting source lines is desired over referencing artificially established numbers attached to listings.

8. Reliability. Since FACES encourages coding practices to increase software reliability, the techniques encouraged by analysis are used in construction of the FACES system. In addition, reliable coding techniques beyond analysis capabilities are employed to detect errors and recover from system malfunctions.

Coding Conventions. To enhance code uniformity and improve transportation of the FACES system, several coding conventions were adopted for implementation.

1. ANSI Standard FORTRAN. ANSI Standard FORTRAN was adopted for system implementation. Where deviations from the Standard were accepted, the potnetial impact of code transportation was carefully weighed by considering features commonlh available from resident compilers. The following are accepted deviations:

   1. Use of IMPLICIT. Since all FACES variables are integer type, the use of IMPLICIT was considered desirable. All routines contain an IMPLICIT (A-Z) statement.

   2. Use of ' in FORMAT character literals and comments. Since the use of ' can be easily converted to the appropriate H form by a mechanical process, the use of ' delimited character strings was permitted in a limited context.

   3. EOF detection for sequential files. ANSI Standard FORTRAN provides no detection mechanism for determining the end of a variable lenght sequential file. Resident machine dependent code is used to determine the end of file. Physical I/O routines are isolated to independent routines to simplify conversion.

Although specified by ANSI standard, allocation of core for multidimensional arrays is not assumed. Additionally, many standard constructions such as EXTERNAL, ASSIGNED GO TO, etc. were excluded from implementation techniques after being judged too difficult to maintain.

2. Variable Names. Variable names are restricted to a maximum of

6 characters. Since all variables are integers, leading character conventions were not considered significant. Where the variable represents a character string, the name chosen is Hxxxx, where xxxx represents the characters. For example, the variable containing the Hollerith character A is called HA.

3. Data Values. In general, FACES variables are either numerical integer values or character strings. Numerical values use positive integers to avoid problems of packed data stored in single words. The value zero is reserved in coding schemes to indicate a null or empty state. If a numerical value is used for a logical indicator, the value zero is used for the FALSE condition and positive value (usually 1) is used for the logical TRUE condition.

Character strings are stored in Hollerith format since this is the only ANSI Standard form of character representation. Character variables also use the value zero to indicate an empty state. Since no machines known to the designers have a valid Hollerith character string represented by all zeroes, this technique should be transportable to new host environments.

4. Physical Computer Words. FACES assumes a minimum integer word size of 32 bits. No more than 32 bits are used by any variable regardless of physical host word size. Bits are numbered for internal use from the leftmost (most significant) starting with the number 0 increasing to N, the right most bit. Characters within a word are numbered from the left most character starting with the number 1 for the first character. At most, four characters are packed in a single integer word.

5. Data Elements. Preference is given to full integer words for holding values. If packing is required, numerical values are packed two to a word. Where the significant value is contained in half an integer word, the data item is called "half word data". Half word data physically occupies a full integer word when unpacked. Machine dependent use of physical half words are not used in FACES.

On some occasoons, bit field flags are used in packing table data. Flag bit fields are allocated to the leftmost position of computer words.

Data values are right justified for numerical codes and left justified for character codes. If the data is held in a half word of host memory, the physical word is divided and the data is justified in the separate regions.

6. Data Structures. Numerous tables are used to implement FACES processing. Tables arc physically implemented by FORTRAN arrays. The array declaration is of the form A(w,l) where w is the width of the table entries and l is the length of the table.

By convention, the declaration of a table requires establishing table control variables. These variables include pointers to access table entries and a length variable to control overflow conditions. If the array name is TAB, conventions require the current pointer to be named to be named PTAB, the pointer to the last entry (for sequential tables) must be named PLTAB, and the physical length indicated by LTAB. Additional descriptive pointers maybe included in the data structure description, however, no firm convention is required

for the additional descriptors. By convention, the table is empty if the pointer to the last entry has value zero.

7. COMMON Block Conventions. Preference is given to numerous small labeled COMMON blocks which are identified with a data structure or processing function. All COMMON declarations are identical in separate routines.

COMMON blocks which implement tables are of the genereal form:

```
COMMON /GEN/ GENTAB(x,y), LGEN, PGEN, PLGEN
```

where

GENTAB is the table space

LGEN is the length of the table (value y)

PGEN is the pointer to the current entry

PLGEN is the pointer to the last nonempty entry in the table

If a hash coded table is constructed, the pointer to the last nonempty entry is replaced by a prime number variable to be used in the hashing.

Where the COMMON block describes an I/O file, the general form is:

```
COMMON /FILE/ FILEFL, RSFILE, LGFILE, EFFILE, PTFILE
```

where

FILE is the file name

FILEFL is the I/O unit number of the file

RSFILE is the file record size

LGFILE is the length of the file

EFFILE is the end of file indicator

PTFILE is the pointer to file records

Not all variables are needed on all files, although they are all declared. If future needs arise for these file descriptors, the assigned names should be used as the required variable.

Blank COMMON is reserved for the largest local table.

8. Statement Labels. All executable statements are labeled in increasing numerical order. Associated code sections are grouped with common leading digits. Substantial breaks in program logic begin new label sequences.

All DO loops are assigned unique terminal labels. All DO loops are terminated by CONTINUE cards to set off the loop. Loop interior code is indented.

FORMAT labels are distinguished from executable labels by conventions.

Where independent cases are treated in the code, statement labels are assigned to indicate the case being processed.

9. Comments. Liberal comments are included with routines. Comments are the primary method of describing what is occurring in the executable code. Comments provide overview, warnings, and assertions of conditions in the program at strategic points.

10. Code Structure. All nonexecutable declarations appear prior to executable code. Declaration of arrays is preferred in the COMMON statement if the array is used by more than one routine.

Code is structured by either statement labels or indentation. All DO loops are indented. Fully indented routines are an experimental form of structured programming applied in FORTRAN. Branches are not

allowed to enter a structured section except at the heading statement. Iterative structures implementing the WHILE condition are pretest loops; UNTIL structures are post-test loops.

FORMAT statements follow the first I/O statement which references the FORMAT. Preference is given to assigning individual FORMATs to each I/O statement.

COMMON blocks are ordered alphabetically on the COMMON label. Routines are ordered alphabetically on the routine name.

11. Error Detection and Control. Substantial code is dedicated to internal error control. Parameters and table entries are checked prior to use. Overflow of arrays is checked. Where errors are detected, error reports are issued. If processing can proceed, corrective action is taken.

12. Programming Techniques. To facilitate modification, physical data structures are concealed from processing routines. Complex structures are managed by independent routines under the direction of processing routines.

Although negative values are avoided in the system, the test for zero is frequently implemented by .LE. or .GE. operations. This technique permits error recovery if an illegal value is assigned.

Routines are coded to be serially reusable. Any values stored between calls are placed in COMMON to prevent problems in future overlay environments. Local variables are not set with DATA statements.

Tables are frequently cleared to zero prior to inserting data.

The clearing operation is protection against possible malfunctions. Processing rotuines do not assume initially cleared tables.

Machine dependent routines are coded as stand alone modules to permit easy modification and transportation. Any routine performing bit manipulations or machine dependent I/O is considered machine dependent.

Integer rounding division is used to compute the integer least upper bound on a value. For example, to compute the number of positions required to store an element of length A in groups of N elements per entry, the computation (A+(N-1))/N is used.

The intrinsic FORTRAN functions MOD, MINO, and MAXO are used in the system. MOD is used to perform table wrap-around searches. Where the table length is L and the current pointer is P, the expression

P = MOD(P,L) + 1

yields the next table position in wrap-around fashion.

13. Initialization. In general, each subsystem is responsible for initializing table entries and data structures unique to the subsystem. If the data structure is used by all subsystems, initialization is performed in BLOCK DATA.

Since the ANSI standard permits DATA statements to set COMMON variables in BLOCK DATA only and restricts Hollerith character strings to parameter lists and DATA STATMENTS, all variables requiring character literals are set in BLOCK DATA.

Data structures and variables not used by all subsystems are established in an INTxxx, where xxx is a subsystem identification.

SPECIAL NOTES: Initial system development permitted heavy use of the two branch logical IF statement. This form was mistakenly assumed to be widely implemented. When it was discovered that the form was restricted to a few machines, statements of the form

```
IF( condition ) 1,2
```

where mechanically converted to

```
IF( condition ) GO TO 1
                GO TO 2
```

Both forms read

```
"IF condition  THEN  1  ELSE  2"
```

VI. Control Driver

Design Considerations

Purpose. The Control Driver coordinates activities of the functional subsystems to produce analysis results requested by the user.

Requirements. The Control Driver manages file positioning and command card interpretation to control FACES activities. The control structure was designed to facilitate operation in an overlay environment with at most one subsystem core resident at a time. Due to limitations encountered in installation, the control was divided into 3 phases of operation, with each phase controlling one subsystem.

The Control Driver should interpret card image control commands given by the user and construct internal control variables to cause proper execution of the desired function. The Control Driver should inform the user of interpretation of the command and actions resulting from the request. Any errors detected should be reported to the user.

In an error environment, a valid subset of operations should be performed. If no valid activity can be derived from the command, no action should be taken.

The Control Driver should isolate subsystems from file positioning problems. Files should be positioned such that immediate use of the file by the intended subsystem is not dependent upon the nature of the run.

Command cards should be easy to construct. Natural language construction is preferred to elaborate syntax. Where formal syntax is

required, the character set and expression format should be similar to FORTRAN constructions.

Strategy. To accommodate overlay execution, the processing subsections are driven from a central routine. The Control Driver is responsible for command card interpretation and file controls, permitting subsystems to be isolated from the activity sequence.

To simplify command card processing, a free field blank delimited format was selected. Each command begins with a keyword followed by qualifiers. The keyword is directly associated with a functional subsystem; qualifiers are selected depending upon the capabilities of individual subsystem controls. A stand-alone command is implemented to initialize the system on the first run.

Since different subsystems require different manipulations and user controls, a link procedure subroutine is supplied to interface each subsystem in the control strategy.

The link process interprets options for a particular phase and establishes control variable values to drive the subsystem in the proscribed fashion.

Accommodating phased operation requires distributing the command control among different job steps of the run. In general, phases are established along the lines of control card order. That is, Phase 1 will accommodate the addition of source code to the system (FFE activity); Phase 2, analysis of code properties (AIR activities); and Phase 3, report production (Report Generator activities),

Control among subsystems is effected by passing the control cards and resident files among the subsystems. Thus, with the exception of

the Control Driver, subsystems are not aware of the phased nature of processing.

Overview of Control Driver Operation

The overview of control envisioned is illustrated in Figure . Required procedures include initial file activities, distinguishing initial runs from follow-on processing, accomplishing user requests, and final file action to permit results to be saved for subsequent runs.

Processing of user requests is implemented as a processing cycle controlled by command cards. The processing cycle is terminated by detecting the end of a command card set.

Control commands are implemented using free format, blank delimited commands. The first entry of the command (command keyword) identifies the subprocess to initiate. Remaining control qualifiers are command card dependent.

Command cards are interpreted as follows:

1. The keyword command item is interpreted to determine the subsystem required for the processing request.
2. A linking procedure is initiated to complete interpretation of the command card and establish file and control variable linkage to the subsystem.

Although command card interpretation is distributed, a common set of subroutine calls is used to access command card data. Command card entries are returned through a single COMMON block.

At the end of commands, file manipulations required to secure the data are performed. Initial and final file manipulations may differ among host machines.

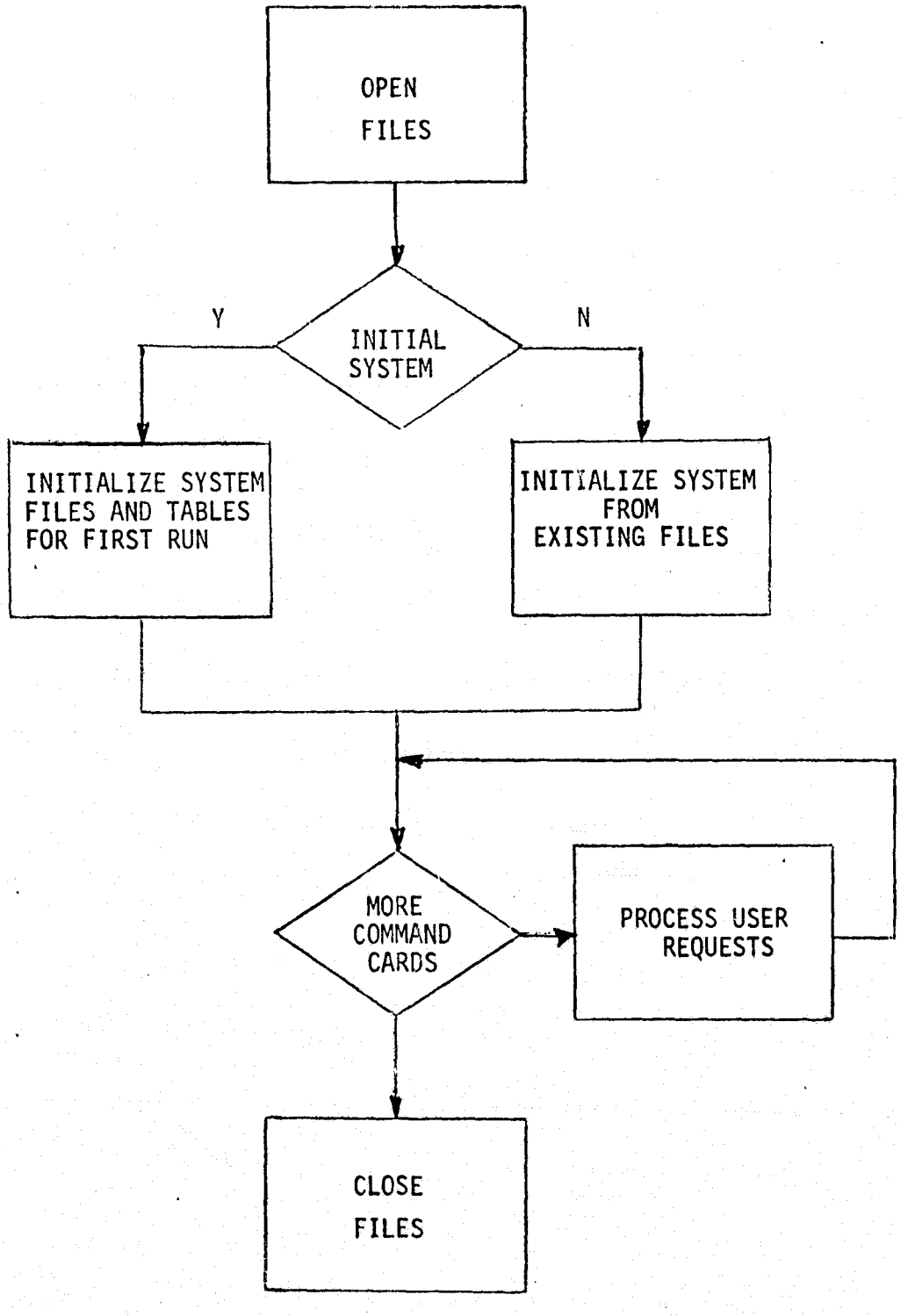

OPEN
FILES
INITIAL
SYSTEM
Y
N
INITIALIZE SYSTEM
FILES AND TABLES
FOR FIRST RUN
INITIALIZE SYSTEM
FROM
EXISTING FILES
MORE
COMMAND
CARDS
PROCESS USER
REQUESTS
CLOSE
FILES

Figure IV-1

Effects of Phased Operation. Although initially designed to operate in overlays, the FACES system is installed in phases. Phased operation became necessary because:

1. A FORTRAN callable sort procedure could not be found and insufficient development resources remained to construct such a routine. Sorting of Flag File entries is required between analysis and report generation.
2. Insufficient time was available to actually set up the overlays among routines.

With an eye toward eventual consolidation of processing, the control structure was adapted to operate in phases.

Phase 1. Phase 1 permits the analysis of FORTRAN source code and incorporation of source in the analysis library.

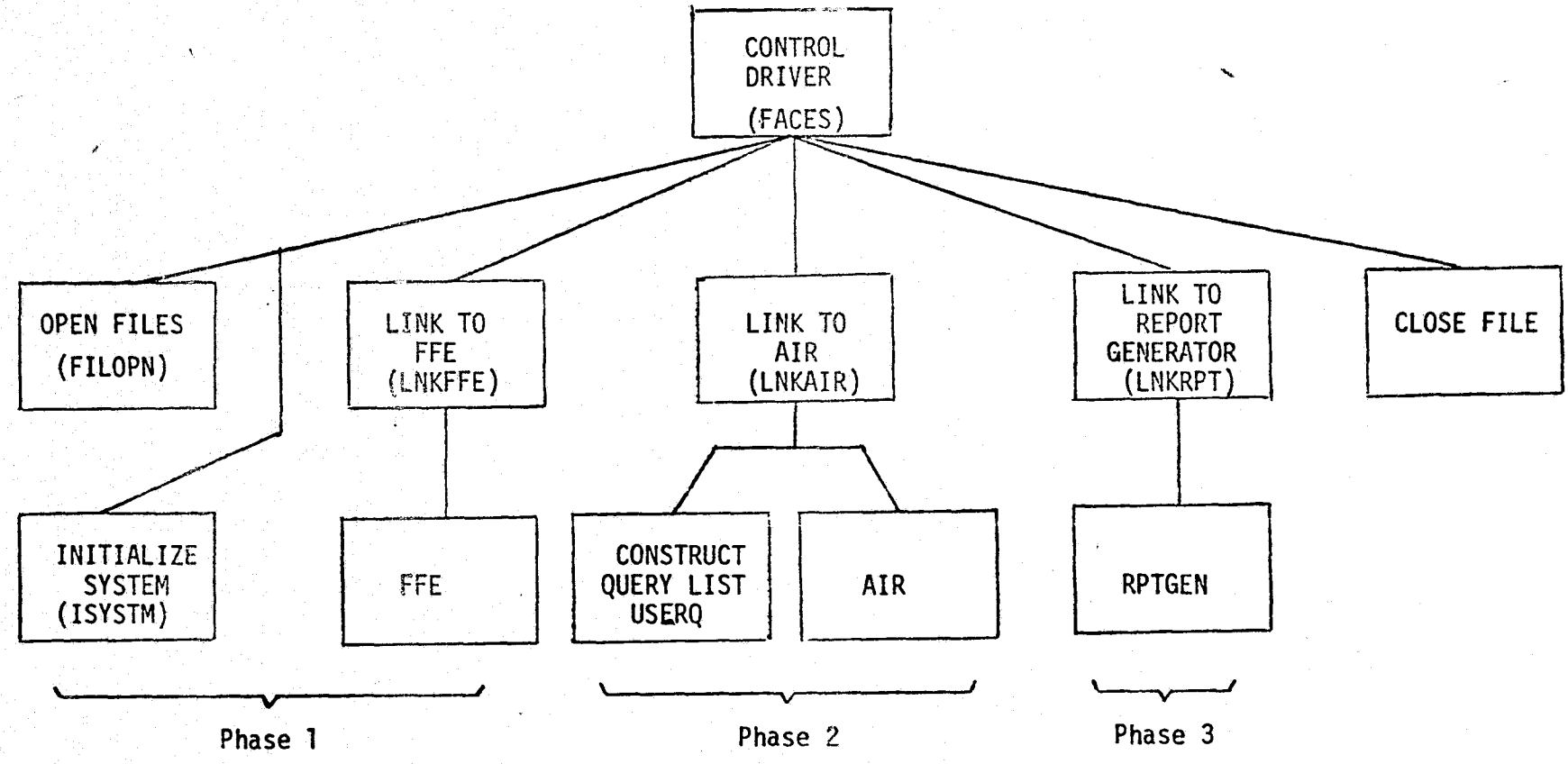
Phase 2. Phase 2 permits the investigation of software in the analysis library.

Phase 3. Phase 3 permits the generation of reports from information gathered during source code analysis.

On each run, all three phases execute. If command cards are absent for a particular phase, no action is taken by that phase. The required sorting of Flag File entries is accomplished between phases 2 and 3 by system software.

The primary effect of phased operation is restriction on command card order and limitation to a single report request on one run.

In implementation, phased operation required adaptation of the Control Driver to the functions permitted within that phase. Essentially, this amounted to reducing the scope of control for three different copies



Control Driver Processing Hierarchy

Figure IV-2

of the Control Driver. Identical control routines are found in each phase. Control routines unique to a given subprocess are not included in phases where that process cannot be performed. For example, the linking routine to AIR is found only in Phase 2.

Processing Command Cards

Command Card Format. Command cards are single card images of 80 columns organized as a free field, order dependent sequence of command items. Command items are delimited by blank characters or special symbols. A series of blanks is equivalent to a single blank.

The first command card item is a command keyword identifying which subprocess to activate for the user request. Remaining card entries are keyword dependent options which may be omitted.

Acquiring Command Card Items. The flow of data from command cards is illustrated in Figure IV-3. The command card entries are scanned in sequential fashion. Groupings of characters are collected by the scan process into "command items".

Command items are one of the following forms:

1. Character strings of one or more alphanumeric characters.
2. Single entries of special characters.
3. Single entries of special termination symbols.

The nature of the command item is established by the first non-blank card character encountered. If the character is an alpha or numeric character, the command card scan extracts subsequent characters

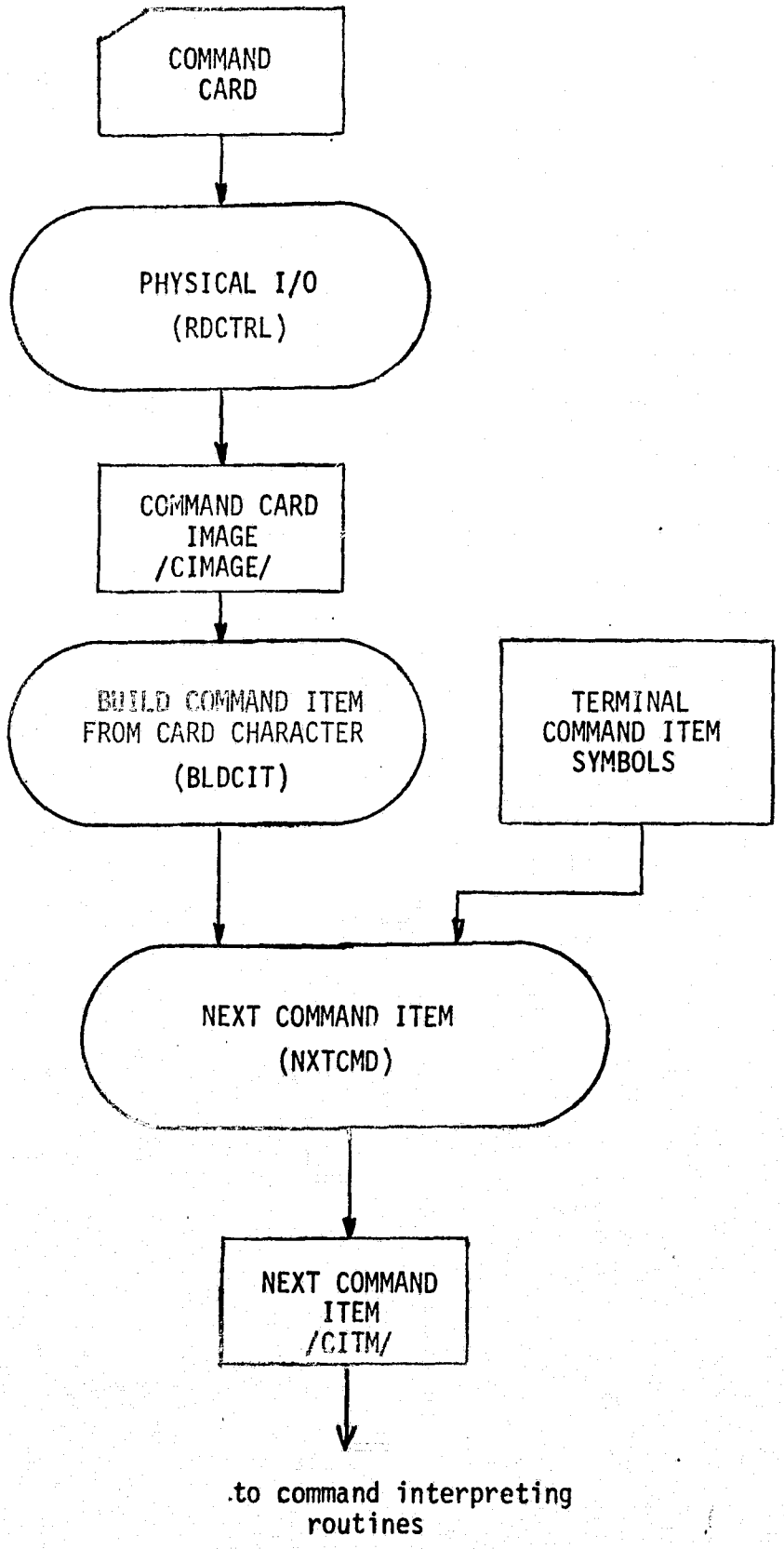

COMMAND CARD
PHYSICAL I/O (RDCTRL)
COMMAND CARD IMAGE /CIMAGE/
BUILD COMMAND ITEM FROM CARD CHARACTER (BLDCIT)
TERMINAL COMMAND ITEM SYMBOLS
NEXT COMMAND ITEM (NXTCMD)
NEXT COMMAND ITEM /CITM/
to command interpreting routines

Figure IV-3

until a special symbol, blank character, or end of card is found. If the first character is a special symbol, a single character item is extracted as a command item.

Special termination symbols are generated to indicate the end of a command card and the end of a command card set (i.e., no more command cards to be processed).

In addition to providing the command item text string, the item is classified by the scan process as: Alphabetic, Numeric, Alphanumeric, Special, or Terminal. This classification simplifies recognition of the command item and assists in detecting and diagnosing keypunch errors on the command.

Where terminal command items are returned (end of command and finish of command cards), the classification symbol is returned both as the classification and as command item text. Establishing the terminal condition as the command text prevents using "left over" data from the last command item in interpreting optional qualifiers.

Command Card Scanning Conventions. Since blanks are ignored except to terminate command items, a pointer convention is required for scanning the card. When a new card is read, the scan pointer is advanced to the first nonblank column. If the card is blank, the initial position of the pointer is beyond the end of the card. As command items are extracted, the scan pointer is advanced to the first nonblank entry after the command item.

When the end of the card is detected, the scan pointer is positioned beyond the end of the card. This state is called an "exhausted card" and indicates the end of command card text.

Detecting Terminal Conditions. When a command item is requested from an exhausted card, an " end of card" command item is returned and the card is set empty (i.e., pointer to last entry set to zero). Acquiring a new card image requires positive acknowledgement of the end of card condition. This acknowledgement is provided by setting the command item empty prior to calling for the next command item.

If command items are requested from empty cards without positive acknowledgement, "end of card" command items are repeated. This control mechanism permits requests for nonexistant options from a card without the danger of prematurely moving to the next card image.

If a request for new card image results in no card being provided, the end of command card set is detected. This condition results in a "finish" code being returned as the command item. The "finish" condition is terminal; no other command item will be returned so long as requests for new data result in no new cards.

Physical I/O of Command Cards. The physical read of command cards results in command card images being read to a COMMON resident card buffer. In addition to reading the card, the nonempty entry pointer is set to the end of the card image to define the character scan region.

If an end of file is detected during the card read, the end of file indicator is set and an empty card (i.e., zero length nonempty card image pointer) is returned. All subsequent card requests are ignored so long as the end of file indicator has not been cleared.

Command Item Access Protocalls. Since the number of command item entries cannot be predicted early in the scan, care must be taken to avoid prematurely reading a new command before the current process is completed. To provide this protection, conventions in access of command card items are established.

Command items are acquired through NXTCMD. The first call to NXTCMD causes the command card to be read. Command items are acquired sequentially from the card image. When command items are exhausted, a series of "end of card" command items are returned. This code is ignored by routines establishing controls for a subsystem; default activities result where end commands are provided.

After the command has been processed, a clearing routine, CMDEND, is called to acknowledge the end of card. If the command card was not completely scanned by the processing routine, CMDEND will flush any unused text, isolating error conditions to a single command card image, and recovering to the next command card.

If a command is unrecognizable, the CMDEND routine will discard the card without need for special processing.

V. FORTRAN FRONT END

Design Considerations:

Purpose. The FORTRAN Front End (FFE) is responsible for producing analysis data from incoming source code.

Requirements. FFE must accept legal FORTRAN code from the user. Input source decks should not require extensive preparation for submission. If the deck contains foreign constructions beyond FACES capabilities, alien code should be ignored in the analysis.

Source code must be analyzed using the rules and interpretations of FORTRAN compilers. The source code should be captured to facilitate result reporting and future system extension. Source code must be correlated with the analysis data produced.

User errors and system limitations should not cause system termination. Rather, analysis should be performed on program modules to the limits of the system's ability. Limitations should be communicated to the user for interpretation of effects.

To accommodate different dialects of FORTRAN, the FFE should isolate where possible, the influence of different target FORTRANS. Machine dependent FORTRAN constructions should be implemented to minimize the effects of different FORTRAN extensions.

Techniques should emphasize transportability of the system to different host environments. Host dependent techniques should be minimized and isolated.

The purpose of FFE is to analyze code, not criticise the techniques used. To the extent possible, deviations in capabilities among dialects should be accepted by the FFE as the broadest capability in known dialects.

Strategy. Since considerable experience was available from FACES Version 1, similar techniques and constructions were applied for Version 2.

Analysis tables are constructed on a module basis to describe the computational activities indicated in the presented program. The source code is captured on a separate file with links to the module data established through table entries. Table generated for a module are recorded on a random file after their creation. File entries are recorded under the module name in a Directory.

Analysis of source code is performed by a blind scan followed by a parser-like analysis of the statement description. Statement entries are used to produce analysis table entries. The scan process uses the rules of FORTRAN to separate elements of the statement text into associated groups of character strings.

Incoming source code is assumed to be legal FORTRAN statements composed primarily of ANSI standard forms. Processing diagnostics are limited to identifying constructions which impede processing by FACES. Many illegal FORTRAN constructions will be accepted by the system without diagnostics; initial correctness is required for generating valid tables.

To accommodate different FORTRAN dialects, code is provided for common extensions to FORTRAN. The presence of the extended form indicates the target FORTRAN is capable of supporting the extension. The absence of

an extended form should not affect interpretation.

Some FORTRAN dialects extend the size limits of variable names and constant specifications. The FFE will accept variable names up to eight characters in length. Numerical constants are limited only by the size of table structures.

Table entries generated should parallel program activities which will occur dynamically in execution. Following tables entries should produce activity profiles consistent with the operation in execution. Where the sequence of operations is not significant in the analysis, arbitrary order is permitted in table entries.

If table space is exhausted in the system, the procedure is to truncate and proceed. Similarly, if unrecognized statements are present, statement processing is abandoned and a valid fragment of statement operation maintained for analysis.

Overview of FORTRAN Front End Operation

To the FORTRAN Front End, processing is a perpetual cycle of presented modules. The highest level control routine manages the break in source code presentation between runs. At the start of processing, the system status is copied to pick up where the last module left off. At the end of processing, status information is saved pursuit to the next set of source code modules.

The FFE is driven from Source Code presented for analysis. Source code text causes various processing paths to be executed. The operation is roughly similar to compilation; however, the FFE is much less formal in analysis.

Emphasis is placed upon extracting the data elements used in the program and the context of their use. Little attention is given to the operational computations specified by the program.

Functional activities of FFE subsystems are roughly identified with the following duties:

1. Scanning - combining source code text into program referenced items.
2. Parsing - interpretation of FORTRAN source lines
3. Table Generation - recording the program specified elements in analysis tables.
4. Error reporting - communication of source code features and processing limitations which may influence the analysis
5. Maintenance support - display operational elements for system tuning and debugging

FFE processing control is constructed around the elements of a FORTRAN program: Module set (FFE), Single Module (PARSER), and single statement (PRSSTM). Analysis tables are emptied at the start of a module, produced during module processing, and saved at the end of the module. During module processing, error messages are issued for the constructions found, and source code is captured for report generation.

Processing a module involves a compiler like operation. A single line of source code (including continuation cards) is acquired and examined by a blind scan processor. The scan uses syntactic and format rules of FORTRAN to associate adjacent characters. Parsing Tables are created

by the scan for analysis by Parsing routines. The Parsing Table entries are a normalized presentation of the source text with unnecessary blanks removed and relevant character strings grouped.

Using the Parsing Table entries, parsing routines process the statement text, making analysis table entries. Program elements and constructions are recorded to support later analysis. Parsing routines are identified with statement types potentially available from FORTRAN and constructions used in these legal statements.

At the end of a module, the generated tables are recorded on a random file and the location of these records and the associated source code is recorded in a Directory. To support recording requirements, modules are identified by name.

Scanning Functions

The Scan process analyzes one FORTRAN statement and produces Parsing Table entries for the statement components. Scanning requires the following activities:

1. Interpretation of FORTRAN card format
2. Distinguish initial lines from continuation lines
3. Distinguish comment lines from source lines
4. Apply FORTRAN syntax rules to associate adjacent card characters as a unit

In addition to the formal rules of FORTRAN, the following possibilities are considered in the scan:

1. Presence of blank cards
2. Comments between continuation cards and/or modules.
3. Potential presence of special directives which are recognized by resident compilers as comments.

Scanning is performed in two phases: 1) Preliminary Scan and 2) Post Process Scan. The Preliminary Scan performs most of the effort in producing the Parsing Tables. In the preliminary process, fundamental character strings are associated and simple compound forms are recognized. The post process activity identifies floating point and complex constant constructions. Additionally, the post process reviews the Parsing Tables for the first zero level equal sign.

Data flow in Scanning. The flow of character text during the preliminary scan illustrated in Figure V-1 To recognize possible continuation cards, a statement ends when the initial line of the next statement is detected.

The scan process examines one statement of FORTRAN text at a time. Card data is acquired from input source code and placed in the source code input buffer. If the card read is an initial line or a continuation of the current line, card text is copied to the Scan Buffer. The Scan Buffer is a normalized concatonation of source code characters with continuation column, label field of continuation cards, and identification field removed from FORTRAN source cards. Comment cards are not transferred for processing.

The contents of the Scan Buffer are analyzed by scan routines. Character strings are associated by scanning rules and accumulated in the

SCAN PROCESS DATA FLOW

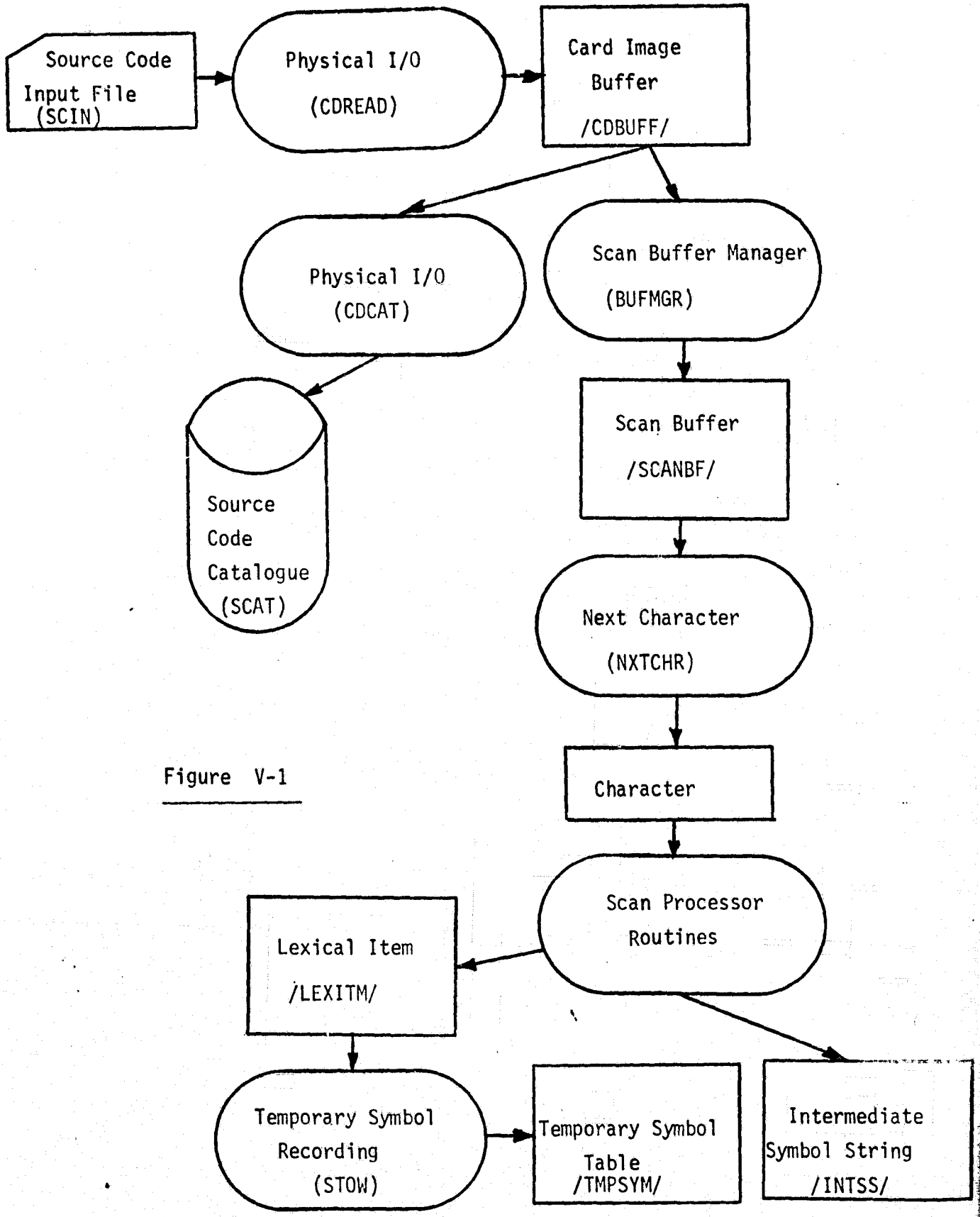


Figure V-1

Lexical Item. Unnecessary blanks are purged. After the collection of individual symbols is complete, the lexical item is stored in the Temporary Symbol Table and an identifying entry inserted in the Intermediate Symbol String. Insertion activity is complete when the statement text is exhausted in the Scan Buffer (i.e., all continuation cards have been processed).

The Scan Buffer is sized to accommodate a moderate number of continuation cards. If the number of continuation cards present requires more data space, the Scan Buffer contents are compressed by purging used data and moving active data to the top of the buffer.

Scan Rules. To accomplish blind scanning, scan rules were developed for associating the character strings of the FORTRAN text. The scan control routine selects a rule to apply based upon the leading character of the next source code symbol. If a multiple symbol string is required, a service routine is called to process the form.

Association rules are developed using "local context" to permit flexible modification of the system. In general, the rules deal with immediately preceding symbols to distinguish one construction from the next.

The principal adversaries of the blind scan are:

1. FORMAT statements. Potential for identifying format component specifications as floating point constants.

2. Procedure calls. Passing a pair of floating point constants may appear to be a complex constant form.

3. Apostrophe Hollerith Strings. The apostrophe may also appear as a separator in direct access I/O.

4. Statement label. Statement labels are processed as a leading integer on the statement.

5. Nondecimal constants. Some nondecimal constant forms cannot be distinguished from variable names. Their use is restricted to constant contexts such as DATA statement constant lists. The blind scan must recognize these as variable names since the context is unknown.

6. FORTRAN keywords. Since blanks have no significance outside literal fields in FORTRAN, keywords (e.g., DIMENSION, GO TO, etc.) are run on with variable names or statement label indicators.

Figure V-2

Scan Rules for Association of Character Strings

Preliminary Scan Process

Routines	Rules
SNALPH	Combine alphanumeric character strings which begin with an alphabetic character. The string is terminated by a special character or the end of card
SNNUMB	Combine decimal digit character strings until nondigit character is found
SNNUMB, SNHOLL	Hollerith forms are indicated by Count H Literal String. The Count may not be a statement label.
SNHOLL	Delimited Hollerith forms are either 'Literal String' "Literal String" The appearance of a double delimit mark indicates a single character in the literal string. To avoid confusion with the ' used as a record indicator in direct access I/O, the character preceeding the ' may not be an operand.
SNNUMB, SNZPRO	Nondecimal constant forms of Count Z Hexidecimal String,Octal String B permitted.
SNZPRO, SNALPH	Nondecimal character strings of O Octal String, Z Hexidecimal String are constants if the total string length exceeds target FORTRAN variable name size. Otherwise, classify as a variable.

SNNUMB	Statement labels are decimal constants which appear first in the statement text.
SNPERD	Logical Operators, Relational Operators and Logical Constants have the form .V. in the preliminary scan. The character string associated with V is one of the known templates for the construction.

Scan Post Processing

Routine	Rule
SIDCPX	Complex constants have the form, (sC1, sC2) where s is an optional sign character C1, C2 are either single or double precision constants. The open and close parenthesis pair must be within at least 7 ISS positions for proper form. If the left parenthesis is preceeded by a V entry, the form is not a complex constant.
SIDFPC	Floating point constants are of one of the following forms M E where M is a mantissa of the form I. or .I or I.I E is an optional exponent of the form E s I or D s I s, an optional sign; I, a digit character string.

If the exponent if specified, the form

I E s I is acceptable.

The form V.I is not a floating point constant.

Treatment of these run on strings must be accomodated by the Parsing routines where context is known.

Character Overrun. Some Scan Rules require a character string to continue until a nonqualified character is encountered. Therefore, in processing a decimal integer, the character after the integer must be obtained before it is know that the decimal integer is over. This condition is called "character overrun." The extra character must be returned to the Scan Buffer for processing the first symbol of the next item.

By convention, the next symbol from the Scan Buffer is always the first symbol of the next lexical item. To correct for the character overrun condition, the Scan Buffer is backed up one position, returning the last character for processing.

End of Statement. The statement is completely scanned when no more continuation cards are found to extend the Scan Buffer contents. Instead of card data, an End of Statement passes the terminal code to the requesting processor. This action will continue indefinitely to neutralize any errors which might ignore the end of statement code. The condition is cleared by setting the Scan Buffer empty to request the initial line of the next statement.

Parsing Table Overflow. The Parsing Table are sized to accomodate FORTRAN statements including a moderate number of continuation cards. If an exceptionally long statement is encountered, the FORTRAN text is truncated to available table space.

The main control routine for the preliminary scan process inspects the Parsing Table status before beginning each new item. If at least

one ISS and TSTAB position remains for FORTRAN items, the process continues. If table space is exhausted, the process is terminated. If a scan service routine requires more than one table position to store the FORTRAN item, the routine is individually responsible for insuring space remains to accomodate the item.

The Parsing Tables are not permitted to be completely filled with source code data. The last few positions are reserved for the end of statement code and a protective buffer of end of statement codes.

If table space is exhausted before the source code is completely processed, the remaining statement text is purged to position the source code to the next statement, and an end of statement code forced into the Parsing Tables. This procedure is called "statement truncation."

Scan Post Process. The Scan Post Process procedure reviews the Parsing Table constructions built by the preliminary scan to collapse compound constructions of floating point and complex constant entries. Each of these constructions is identified by a scan rule routine and collapsed by a separate routine.

In addition, the post process procedure identifies the Parsing Table position containing the first zero level equal sign, if one exists. A zero level equal sign is an equal sign not enclosed in parenthesis. This usually indicates the presence of special FORTRAN statements including assignments, DO's, etc.

Source Code Cataloguing

Since Comment cards are discarded by the scan process, the source code catalogue is constructed in the scan phase (routine BUFMGR). As

source code lines are read from the input file, the card image is written to the next sequential position of the SCAT file. Card images are cataloqued by relative card counters which are reset at the end of each module.

As source lines arrive, the first and last card numbers of a FORTRAN source line are recorded. Comment cards between lines are ignored in the counts. Thus, the last card of one statement may be separated by more than one value from the first card of the next statement. If comments appear between continuation cards, the comment is catalogued as part of the statement.

If a FORTRAN statement contains two logical portions, such as a logical IF statement, both portions will have the same card image counts.

The card of the statement is the module relative card image of the initial line. The initial line is also the last card if no continuation cards are found. If continuation cards are detected, the last card is the final continuation card in the set.

Statement Parsing

Statement parsing is similar to a compiler interpretation of FORTRAN syntax. The FFE is interested in extracting the data elements and context of their use for analysis; actual mathematical munipulation of program elements is ignored. During the Parse, local tables are constructed. In comparison to compiler operation, Symbol Table entries are similar to core allocation for program variables; use table entries, object code; and transition pairs table entries, transfer vector.

The parsing procedure is basically a single pass left to right parse on elements of the Parsing Table. The analysis proceeds using bottom up production on elements found in a statement. The parser is looking for a program composed of a header statement, followed by a (possibly empty) set of body statements, terminated by an END statement. If a new module header is found before the END, a premature header card is assumed indicating the absence of an END card on the current module.

An overview of the parsing activity is shown in Figure V-3. Control is divided into module level control and statement level control. The statement parsing routine requests card data to be placed in the Parsing Tables. Statement lables are processed by the statement parsing routine, advancing the Parsing Table to the first statement text entry. The primary parsing decision is made from the zero level equal sign indicator. If the statement contains a zero level equal sign, control passes to a decision routine for those statement forms. Otherwise, the statement is processed by a FORTRAN keyword which appears as the first entry of source code text.

Zero Level Equal Process. The zero level equal sign statement process must distinguish between the statement types of:

1. Assignment Statements
2. Statement Function Definitions
3. Do Statements
4. IF statements with a logical assignment statement. In the process, the statement type is established for the current statement. Control then passes to the appropriate statement processor.

Key Word Process. The keyword process uses the first entry of the statement to determine the type of FORTRAN statement to process.

Parsing Control Overview

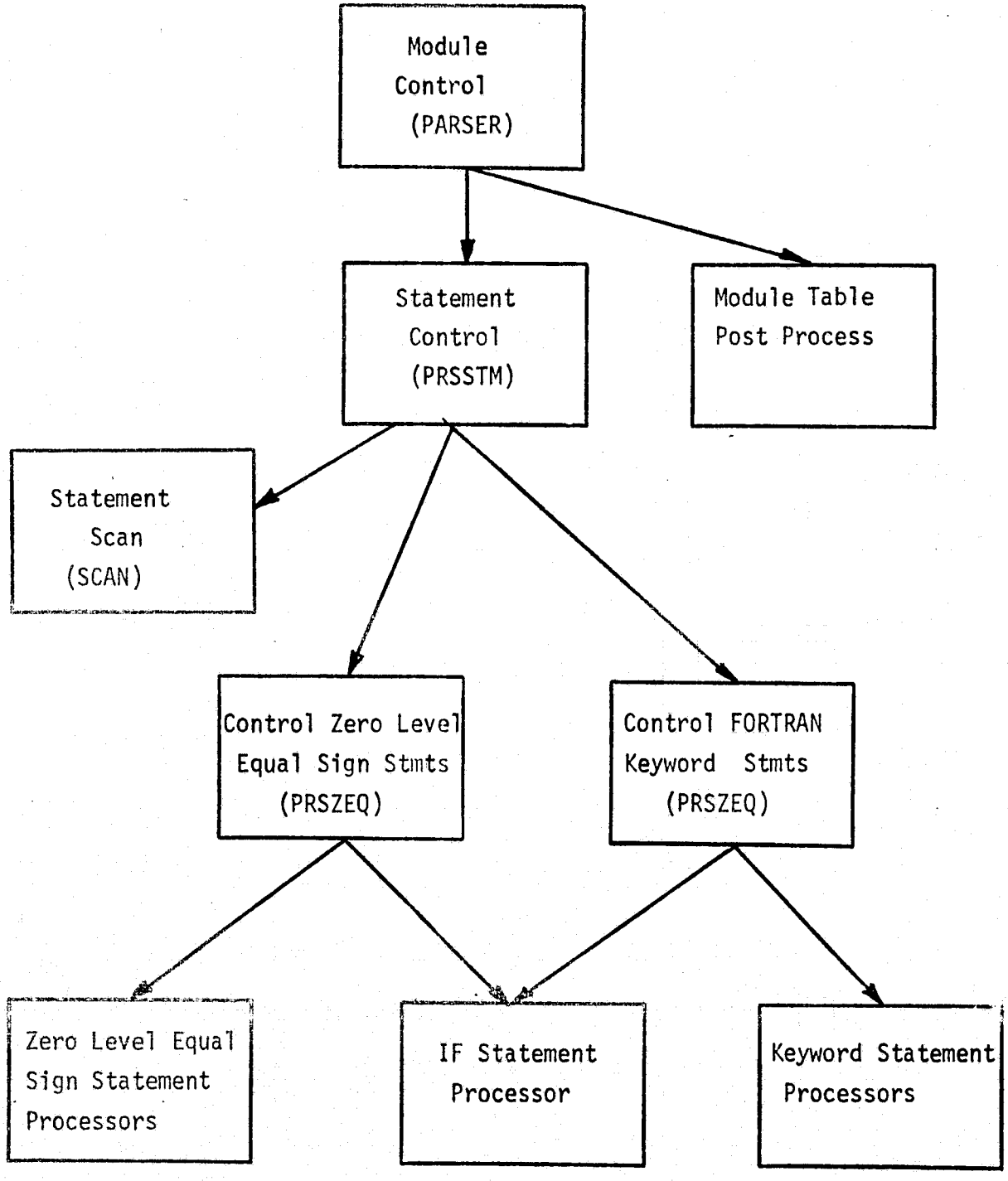


Figure V-3

The keyword is identified by the leading four characters of the keyword. If a known statement is detected, the statement type of the current statement is established and control passed to the appropriate statement processor.

Parsing techniques and conventions. In processing FORTRAN keywords, correct spelling is assumed. This assumption arises from the correct code assumption central to the FACES strategy. If the word is improperly spelled, correct operation will usually result if the character count is at least correct. FORTRAN keywords are not reserved words. Therefore, parsing routines must be aware that the programmer may use keyword character strings as variable and subprogram names.

Parsing Table entries may be run on by the blind scan. In processing keyword constructions, this condition must be corrected. For example the construction,

```
DIMENSION VAR
```

will appear as one continuous string in the Parsing Table entries. The character string VAR must be extracted from the leading keyword symbols DIMENSION. This action is normally performed using a local holding array and the routine SHIFTY.

Once the variable name is extracted, the Parsing Table entries are modified to normalize the appearance of structural elements in the statement. An alternative is to directly treat the extracted name and advance the Parsing Tables past the run on entry.

Support routines are used to process major subconstructions of FORTRAN statements. Each routine processes Parsing Table entries until

the construction is completely recognized or a foreign form is encountered. Control is returned to the calling routine with the Parsing Tables positioned to the last entry not processed.

Parsing Table Manipulations. Since the Parsing Tables are composed of a synchronized table pair with implied association, there is substantial danger of lost control through table positioning error. For this reason, modification and radical manipulation of table entries are avoided where possible. By convention, the tables are always maintained in a synchornized state. This convention carries over even to error processing.

In some instances, the Parsing Tables must be modified by deletion or replacement of entries. Routines are provided to perform this service and centralize modification.

Generally, once an element has been passed over, there is no need to return. On some occasions, hoever, the tables must be restored to a different position. ISS is the master control for this movement. Parsing table positions are recorded in terms of the ISS position. The TSTAB position corresponding to an ISS entry can be developed given any properly synchronized state. A service routine, FNDTST, is provided to determine the proper TSTAB entry position.

In repositioning the Parsing Tables, the TSTAB position must be determined before any adjustment is made to the pointers; current pointer values are used to determine the proper new position.

Recognition of the end of statement code is reserved for the highest level routine. To lower level routines, the end of statement

code is simply an unknown symbol. The end of statement code is constructed to not match any template used in the searching process.

Although correct code is assumed, incorrect code is not permitted to drive the system into an error state. For this reason, parsing routines frequently look for the end of a sequence rather than a specific symbol. For example, in a DATA statement, the end of a comma series is used to terminate a loop rather than the final "/". If the "/" was missing, a search loop keyed on the "/" would not terminate.

At the end of each statement processor, a call is made to the end of statement routine. This is in effect an assertion that the statement has been completely processed. If the statement was only partially processed, or superfluous text remains, this routine will report that some text was not processed. This condition might result if a foreign construction was found during statement processing.

Note that the logical IF statement construction containing a conditional statement does not call the end of statement process. Rather, this construction is treated as two statements. After processing the IF condition, the Parsing Tables are left positioned to the first entry of the conditional statement. The second portion is processed independently on the next cycle.

Statement Abortion. When an unrecognized construction is encountered in the Parsing Table which inhibits further processing, the statement in progress is aborted. Abortion does not affect the tables entries recorded prior to encountering the problem or necessarily inhibit further analysis. In effect, abortion simply acts to terminate the

card text prematurely.

Abortion is accomplished by simply positioning the Parsing Tables to the end of statement text. The analysis routines react as if the text was prematurely exhausted. An error message is issued to indicate where the analysis was terminated.

Special Notes on Parsing. While the great majority of parsing is straight forward, some special mention should be made on unusual problems.

1. Parenthesis Processing. Parenthesis balancing is accomplished on a local level. In general, a routine which processes the opening left parenthesis is responsible for matching the balancing right parenthesis. No global count is maintained for parenthesis processing. Parenthesis counts range from the value 0 (balanced parenthesis) upward to a positive count. The count is incremented on left parenthesis and reduced on right parenthesis. An unbalanced right parenthesis is treated as a return condition in service routines.

2. Arithmetic Expressions. Processing arithmetic expressions is complicated by the possible use of subexpressions as array subscripts or function parameters. Since this is basically a recursive process, allowances must be made to process the recursive form in a nonrecursive FORTRAN environment. To accomodate this requirement, arithmetic expressions are processed as simple forms up to a potential candidate which might contain a subexpression as a component. Simple arithmetic expressions are defined as expressions which contain an array reference with simple operand subscripts as the most complex component.

When an array or function reference is encountered, simple arithmetic processing is interrupted and the subscripts or actual parameters examined. If complex forms are discovered, the subscript or parameter is replaced with a temporary variable to produce a simple operand form. Function references are processed before returning to the simple arithmetic expression process. The functions name is left in the Parsing Tables to be recognized as a simple variable upon return to simple expression processing.

Notice that the simple arithmetic expression process is used for both the exterior arithmetic expression and the subexpression process. For this reason, counters and indicators used by the simple arithmetic expression process must be carried on the calling routine side of the interface. Otherwise, routine sharing could not be used.

3. Assignment Statements. The arithmetic expression of assignment statements is processed first. This approach is taken to permit Use table entries to parallel the reference pattern of compiled code. If the assigned variable were processed first, an "output" Use would appear before the "input" Uses in the table. This could lead to incorrect analysis.

Processing is accomplished by moving the Parsing Tables to the right of the equal sign, processing the expression, then returning to the assignment variable. Notice that the assignment variable might be a conditional statement in a logical IF construction.

Table Production

A principle activity of the FFE is to produce analysis tables

depicting code interactions. These include both the Local Tables of a module and Directory entries for accessing the tables and source code.

Symbol Table. Symbol Table entries are made by the parsing routines as source code is processed. Since FORTRAN permits implied definition of variables, the Symbol Table is not complete until the last line of FORTRAN source has been processed. As a result, each variable and constant processed is treated by the parsing routines as a new symbol. The character string is passed for recording in the Symbol Table. If the Symbol is already present, the table is positioned to the current entry; otherwise, the symbol is added to the table.

Use Table. The Use table entries are recorded as source code is processed by the parsing routines. The Use table is constructed in a sequential fashion. Since Uses are associated with recorded symbols, the Symbol Table must be properly positioned before the Use is recorded. The Use is attached to a list of references for the Symbol currently selected in the Symbol Table.

Node Table. Node Table entries are made at the end of each statement. The entry is made using the current statement description accumulated by the parsing routines. Graphical entries are constructed after the module has been completely processed.

Graphical transitions are recoreded in the Transition Pairs Table as program branches and boundary conditions are detected in the parse. The transition Pairs table is a temporary structure which is cleared at the start of each module. Since statement label references may be made to labels which are defined by later statements, no attempt is

made to identify defined labels until the module is complete. At completion, the Symbol Table and Use table entries are used to determine node numbers of specified branch references.

Predecessor/Successor Tables. Predecessor and Successor tables are constructed by the same module post processing procedure used to create graphical entries in the Node table. The converted Transition Pairs Table entries are used to establish "explicit" transitions. Implied "next statement" transitions are produced by examining adjacent entries of the Node Table. Current graph production techniques include non-executable statements in the program graph.

Local Table Recording Techniques. To establish Symbol Table entries, the type and class code for each entry must be determined. In general, the highest level control routine may dictate the type and/or class; alternatively, the type and/or class may be defaulted. If the code is dictated, a positive integer value is passed by parameter; if the code is defaulted, the value zero is passed.

A variety of methods may be used to establish the proper type and class codes. Once a positive code has been assigned, the code is maintained. The type and/or class may be derived from a statement context use (e.g., the name of a subroutine, an entry in a type statement, etc.), implied from the ISS code of the operand (see Table V-4), or determined from the character string during Symbol Table insertion (e.g., a variable name).

Since symbols may change type and/or class characteristics between the initial appearance in the source code and a later reference,

Parsing Table Entries

Intermediate Symbol String (ISS)	Temporary Symbol Table (TSTAB, TSTOVR)	Implied Type	Implied Class
V	8 characters of alphanumeric text		Scalar Variable
I	String of numeric digits	Integer	Constant
F	Character string of a floating point constant	Floating Point	Constant
D	Character string of a double precision floating point constant	Double Precision	Constant
C	Character string of a complex constant	Complex	Constant
H	Character string of a Hollerith constant	Hollerith	Constant
T	Character string of a logical constant	Logical	Constant

Table V-4

a special routine, AMBSYM, is provided to resolve ambiguities. The ambiguity process also distinguishes between symbols which have the same character string but are not equivalent symbols (e.g., the statement label 10 and the logical constant 10).

To isolate parsing routines from table structure, table manipulations are normally performed by interface searching routines. These routines are implemented as logical or integer Functions. Given a specification, the table contents are searched. If the required entry is found, the table pointer is positioned to the matching entry. Note that positioning the table requires modifying a COMMON variable; therefore, these routines have "side effects". The routines are constructed, however, to yield repeatable results given the same input conditions and table contents. Thus, they should not result in improper optimization.

Special Cases for Local Table Production. While the general approach suffices for the vast majority of cases, some constructions require special treatment.

1. External Routine Names. When a routine name is passed by parameter to another routine, the name must appear as an EXTERNAL statement entry. When the entry is detected in processing the EXTERNAL statement, a class code of "external" is assigned to the symbol. Since it is unknown whether the routine is a subroutine or function until it is used, a type code is assigned based upon the symbol name, If it is later used in a subroutine context, the type is reversed to "neutral".

On the other side of the interface (i.e., the called routine), the subprogram name is passed by parameter. The character string used in

the subprogram may not be the same as the routine's actual name. Therefore, if the name referenced as a function or subroutine is passed as a parameter, the class code of "external" should also be assigned.

2. Arrays. Arrays may appear in FORTRAN statements without subscripts. Therefore, all variable names are processed as scalar variables except where array declarations are possible. Since arrays and scalar variables must have different names, the ambiguity process recognizes scalar variable references to be references to arrays.

3. Statement Function Dummy Parameters. Statement function dummy parameters may have the same character string as a program variable. This will cause problems if the program variable is declared on a card preceding the statement function definition. Current ambiguity rules permit linking scalar variable references to declared dummy parameters, however, this technique will also link all previous references to the same character string. This error should be corrected in future versions.

Table Overflow. Since FORTRAN only permits fixed length tables, some exceptionally long programs may cause table overflow. If a table overflows, the recording routine is responsible for not addressing the array out of bounds. Normally, the entry is simply discarded and processing continues.

Continuation of normal processing is used to flush remaining text not processed. Error disgnostics are issued to inform the user of the overflow situation.

Constructing Directory Entries. Directory entries are recorded from the completed Symbol Table of a processed module. The module

name assigned is recorded along with access information to the Table File and Source Code Catalogue. Any secondary entry points are recorded in the Directory with the same access information as the primary entry point. References made to other modules are recorded as Directory reference entries. Care is taken in references to avoid entering character strings which are not actual module names (e.g., passed by parameter).

FFE Error Reporting

Two types of Errors are reported by the FFE:

1. System Anomolies. Unusual conditions encountered during processing which might indicate a system malfunction.

2. User Message Reports. Limitations or conditions which affect analysis table construction.

FFE Anomolies are reported by calling the anomoly routine. The calling routine passes identification information indicating the detecting routine and condition which prompted the report. The anomoly routine combines the reported information with the current module and card number to issue the printed report message.

User messages are accomplished by first provided message data in the error message common data structure. By convention, the message data may be passed empty if a single line error message is to be constructed. Flag identification is passed by parameter to the message writing routine. Flag identification information is combined with the common data passed to enter records on the Flag File. Card numbers for attaching the message are acquired from the current statement.

Note that the error reporting routines service the scan, parse, and

post functions. Since the card number differs in these routines, an approximation is made based upon card counts of the various phases.

Maintenance Support Features.

Since the FFE does not normally produce printed output, maintenance trace features were kept in the production version. These features allow printing source code as it is analyzed, dumping the contents of the Parsing Tables constructed for each statement, displaying the contents of tables constructed and collecting statistics on table usage.

The maintenance features are activated by setting variables in COMMON block /FFEOPT/ through modification of the BLOCK DATA routine.

VI. Automatic Interrogation Routine (AIR)

The Automatic Interrogation Routine traverses the tables produced by the FORTRAN Front End (FFE) in search of incongruous FORTRAN language constructions. The user has control over which type of incongruous language constructs AIR searches for, as well as some control over the output format of some of these constructions discovered by AIR.

AIR has the ability to search for any combination of the following unusual language constructions:

1) ANSI Standards function names not used as an ANSI Standards function.
2) FORTRAN "reserved" words not used as a FORTRAN reserved word.
3) Data statements not in BLOCK DATA referencing COMMON Block variables.
4) Function parameters assigned values within the function itself.
5) Multiple branching statements (such as the arithmetic IF) which do not branch to the statement immediately following them.
6) DO Loop control variables assigned values within the loop itself.
7) DO Loop index variables used after the DO Loop has terminated under normal conditions.
8) Local variables assigned values but never used.
9) Uninitialized local variables.
10) COMMON Block misalignment.
    a) Corresponding COMMON Block declarations which do not have the same number of entries.
    b) Corresponding entries in corresponding COMMON Block declarations which do not have identical types.

c) Corresponding entries in corresponding COMMON Block declarations which do not have identical dimensions.

d) COMMON Blocks which appear in only one module.

e) Corresponding entries in corresponding COMMON Block declarations which do not have identical names.

f) Corresponding entries in corresponding COMMON Block declarations which do not have the same size.

g) Corresponding COMMON Block declarations which do not have the same total size.

11) Parameter List misalignment.

a) Corresponding parameter lists which do not have the same number of entries.

b) Corresponding entries in corresponding parameter lists which do not have identical type.

c) Corresponding entries in corresponding parameter lists which do not have compatible dimensions.

12) Cyclic calling sequences.

A query is defined as a search for one of the above constructions.

The user has some control over how certain unusual language constructions are to appear in the report on the software system evaluated. The user may specify that a message is to appear within the source code listing immediately after the violation and/or after the source code listing.

A user may exercise this control over the following queries:

1) Search for a DO Loop index variable used after the DO Loop terminated under normal conditions.
2) Uninitialized local variable search.
3) Any of the COMMON Block misalignment searches.
4) Any of the Parameter List misalignment searches.

Operational Overview of the AIR Subsystem

The AIR subsystem is a three-level hierarchy.

The first level consists of the driver of the AIR system, the routine AIR itself. It invokes those routines which satisfy user requests and FACES system requests. FACES system requests are satisfied before any user requests can be processed. User requests consist of queries and the requests for certain report formats. FACES system requests consist of requiring certain global tables to reside in main memory.

The second level consists of the queries themselves, routines to create certain global tables, and routines to move these global tables from main memory to secondary storage and vice versa.

The third level consists of utilities. These utilities are the heart of the AIR subsystem. They are the only routines that can directly access the tables built by the FORTRAN Front End.

Before AIR can execute, two other processes must already have been completed.

1. The FORTRAN Front End must have already parsed the source code which is to be examined and packed this information into various tables.
2. The driver of the entire system (FACES) must have already placed all system and user requests into list which is accessible to AIR.

After the above requirements have been met, AIR examines the list sequentially, satisfying system and user requests as the list is

traversed. After the list has been completely traversed, control is returned to FACES.

As user requests are processed, AIR searches for specified unusual language constructions. Whenever one of those language constructions is located, warning messages are sequentially written to the Flag File.

AIR Abbreviations

The following abbreviations are used in AIR:

I Data Structures

'COM' = COMMON Block Name Table

'DIR' = Directory

'IS' = Inverse System Hierarchy Table

'ISD' = Inverse System Hierarchy to Directory Table

'LIN' = Linked List Table

'LIS' = List Table

'MAP' = List Table Map

'NOD' = Node Table

'PRE' = Predecessor Table

'SH' = System Hierarchy Table

'SHD' = System Hierarchy to Directory Table

'STK' = Control Stack

'SUC' = Successor Table

'SYM' = Symbol Table

'USE1' = Linked Usage Table

'USE2' = Statement Number List Usage Table

II Types

'A' = Alphanumeric

'I' = Integer

III Scalars

'L' = Length

'P' = Current Row Pointer

'PLE' = Pointer to Last non-empty Row

'PRM' = Prime Number

AIR Basic Search Technique

The FFE parses the source code submitted by the user and produces a large data basa, which is organized in various types of intra- and inter-connected lists in tables. AIR then searches through these lists, searching for patterns determined by the user or the system. A successful pattern search can be thought of as a template match.

The basic approach for any pattern search begins with AIR traversing a list. The list is traversed until an element of the list is found which matches part of the template. The search then enters another list. The entry point into the second list is determined by the location in the first list which contains the template matching element. The search then enters a third list, the entry point being determined by the second list.

This chain of operations continues until one of the following occurs:

1. The entire pattern has been matched. AIR then takes some appropriate action, such as setting warning flags if the investigation is searching for some incongruous language construction, and then continues the search looking for other occurrences of the same language construction.
2. The n^{th} list has been traversed to its end. The search returns to the location in list n-1 that was last being investigated, and the search continues as before. If the n^{th} list is the 1^{st} list, i.e., the initial list, then the search has been completed.

Example:

Search for elements which are 'A' or 'B' and which are '3', i.e., elements which satisfy the logical expression (AvB)^3.

Table 1

Index	Elements	Pointer to List in Table 2
1	V	1
2	A	0
3	A	3
4	end of list	

Table 2

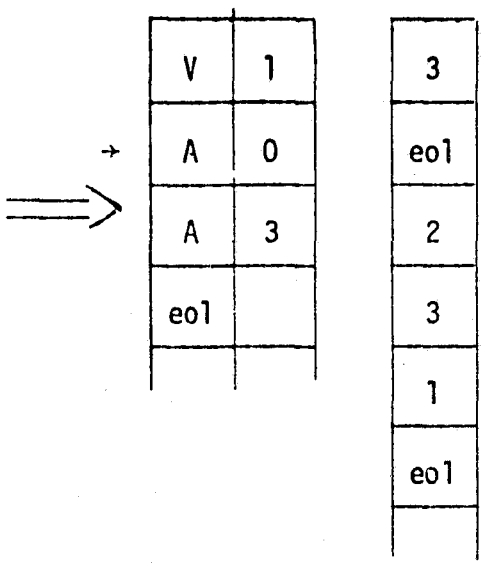
Index	Elements
1	3
2	end of list
3	2
4	3
5	1
6	end of list

Start searching through the list in Table 1 ⟹

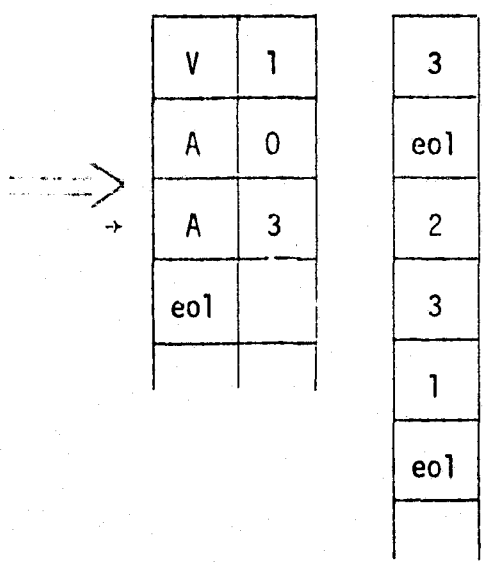
V	1
A	0
A	3
eol	

3
eol
2
3
1
eol

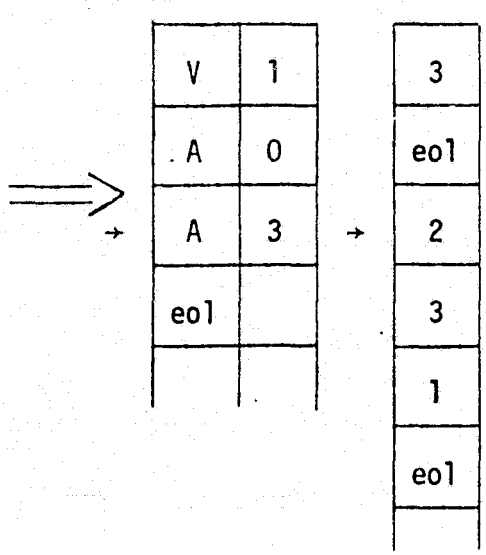
The first element of the first list is not an 'A' or 'B'. Goto the next element in the list.



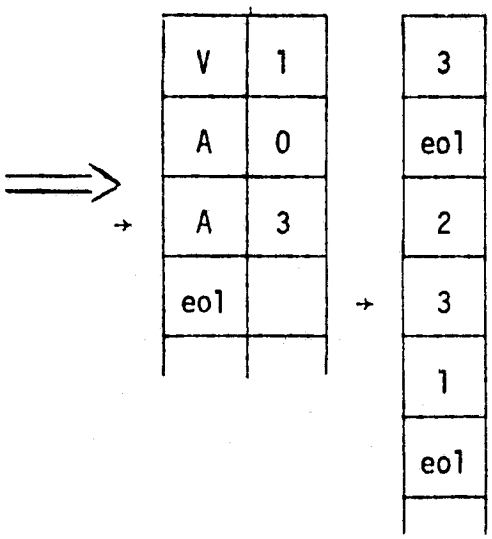
This element is an 'A'. This represents a partial template match. But it has no entries in the second list. This is the same as saying that it does have a list in Table 2, but that list has been completely traversed. Goto the next element in Table 1.



This Table 1 entry represents a partial template match. Start examining the corresponding list in Table 2.

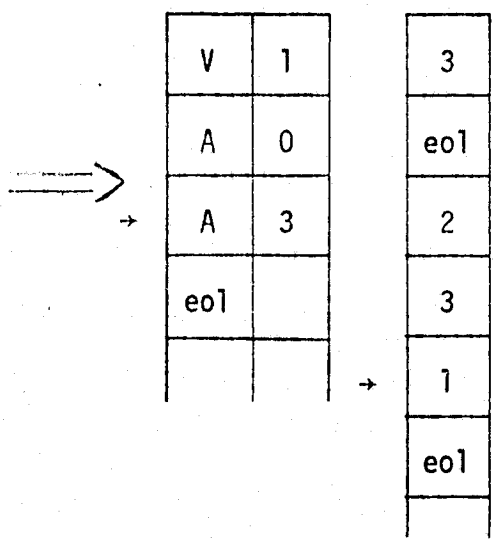


The first element of the list in Table 2 is not a '3'. Continue traversing the list.

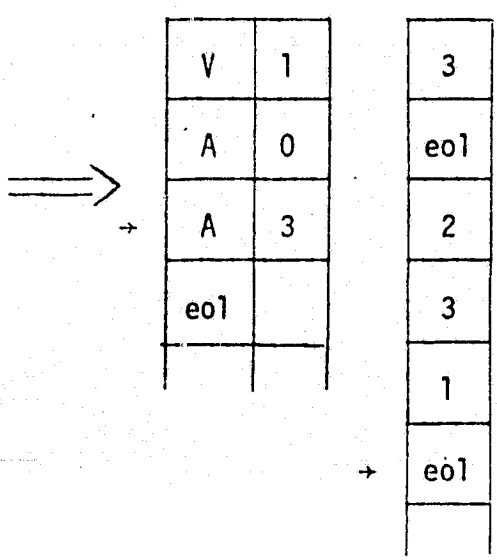


The second element of the list in Table 2 is a '3'. A complete template match has been found. Output a message.

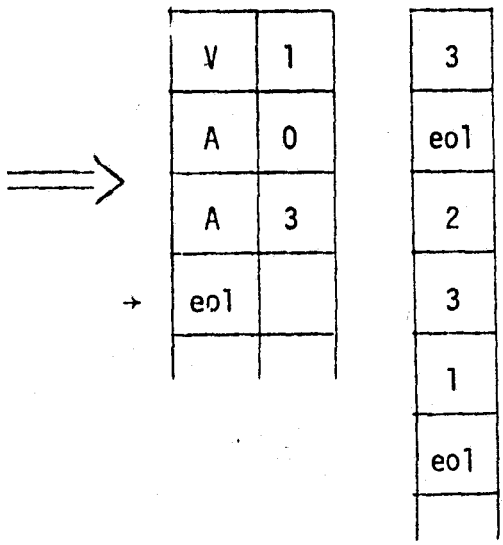
Now search for other occurrences of the same pattern. Continue traversing the list in Table 2.



The third element in this list is not a '3'. Continue traversing the list.



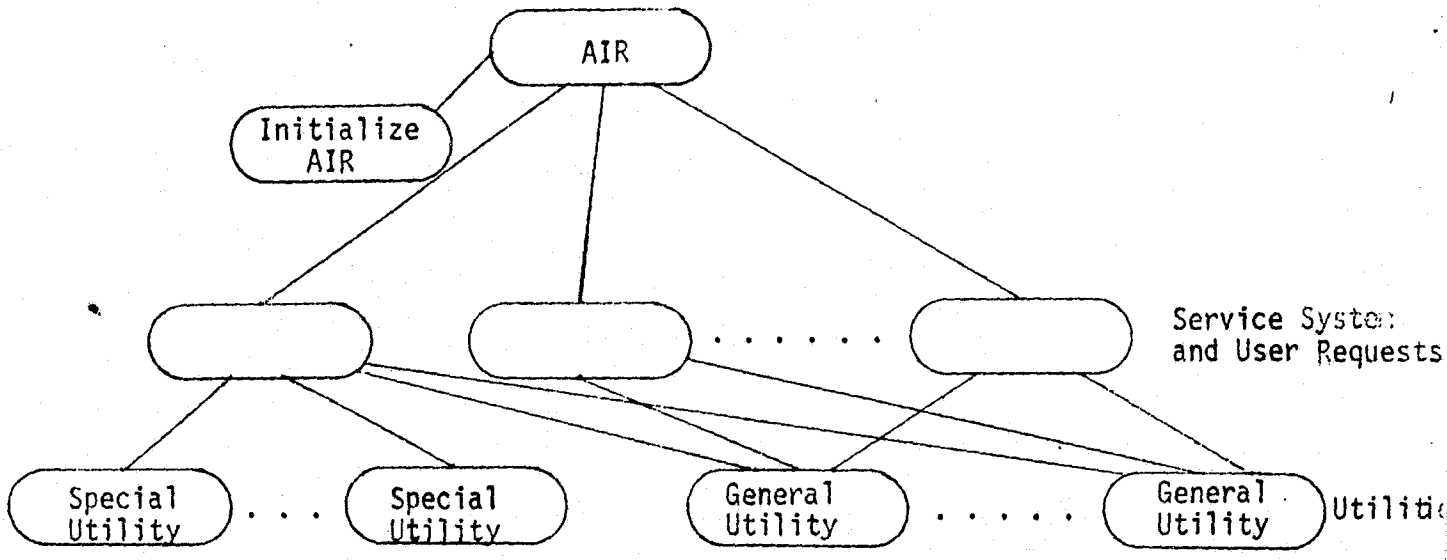
The list in Table 2 has been completely traversed. Return the search to the list in Table 1.



The list in Table 1 has been completely traversed. The search has been completed. All elements which are 'A' or 'B' and which are '3' have been located.

Conceptual Hierarchy of AIR

AIR can best be conceptualized as a three-level hierarchy. The first level consists of the driver of the AIR subsystem and a routine to initialize the AIR subsystem. The second level consists of routines which service system and user requests. The third level consists of utilities; this category can be broken down into general purpose utilities and special purpose utilities.



AIR's calling hierarchy is different from AIR's conceptual hierarchy.

Pattern Searches

As discussed in the section entitled, "AIR Basic Search Technique", pattern searches examine lists.

There are four basic components of the AIR subsystem which are involved with traversing lists. They are

1. The Control Stack
2. The 'Initial Entry' subroutine
3. The 'Table to Table Transition' subroutine
4. The Forward-Backward Register

The Control Stack keeps track of which lists are being traversed (examined) for a specific pattern search, and exactly where a traversal is within a given list. The Control Stack contains four pieces of information for each list being traversed.

1. The module number of the module which contains the list.
2. The name of the table which contains the list.
3. The list indicator for the list. If the list indicator is greater than zero, the list has not been completely traversed. If it is equal to zero, the list has been completely traversed.
4. A pointer to the location in the list indicating exactly where the traversal is within the list.

For a more detailed description of this data structure, see the discussion on the Control Stack.

The top of the Control Stack refers to the list currently being examined. As a new list is examined, information concerning it is placed on the top of the Control Stack. After a list has been completely traversed, its associated information is removed from the top of the Control Stack. Adding information concerning a new list to the Control Stack is called a 'push'. Removing this information is called a 'pop'.

The Initial Entry subroutine deals only with sequential lists whose lengths are equal to those of the tables in which they reside. Traversing one of these lists means that every row in the table within which the list resides is examined.

When this subroutine is called, one of the following occurs.

1. If the Forward-Backward Register indicates 'forward', then appropriate information is added to the top of the Control Stack (a 'push'), with the pointer pointing to the first row in the table.
2. If the Forward-Backward Register indicates 'backward', then the information at the top of the Control Stack is updated to refer to the next element of the list. If there is no next element, i.e., the list has been completely traversed, then the Control Stack is 'popped', and a flag is set.

The Table to Table Transition subroutine deals with lists whose entry points are determined by other lists, i.e., the location of the first element of the list is indicated by some other list. The two lists are always in different tables. Thus, there is always a transition from one table to another.

When this subroutine is called, one of the following occurs.

1. If the Forward-Backward Register indicates 'forward', then appropriate information is added to the top of the Control Stack (a 'push'), with the pointer pointing to the first element in the new list. If for some reason the transition to the new list cannot be made, then no new information is added to the top of the Control Stack, and a flag is set.
2. If the Forward-Backward Register indicates 'backward', then the top of the Control Stack is updated to refer to the next element of the list. If there is no next element, the Control Stack is 'popped', and a flag is set.

The Forward-Backward Register determines what occurs in a number of subroutines, including the Initial Entry subroutine and the Table to Table Transition subroutine. Except for the routine AIR, the Forward-Backward Register always indicates the direction of the flow of control through an AIR subroutine. If the flow is going forward, then this register is set to 'forward'. If the subroutine has just executed a backward branch, then this register is set to 'backward'.

Occasionally, it is necessary to add an entry to the Control Stack without calling IE or TT. This can be done with a call the the PUSH subroutine.

Often, it is either convenient or necessary to remove an entry from the Control Stack without calling IE or TT. This can be done with a call to the POP subroutine.

Example: Search for scalars used as input variables in assignment statements in module number 3. Assume the AIR subsystem has been initialized and that module 3 already resides in main memory.

Statement Number	Source Code
1	SUBROUTINE FLAG
2	IMPLICIT INTEGER (A-Z)
3	A = A + 20
4	RETURN
5	END

Symbol Table (abbreviated)

index	Symbol	Class	Use Pointer
12	FLAG	Subroutine	1
196	20	Constant	3
322	A	Scalar	2

Use Table (abbreviated)

index	Statement Number	Usage	Forward Pointer
1	1	Declaration	0
2	3	input in assignment statement	4
3	3	input in assignment statement	0
4	3	output in assignment statement	0

The algorithm would be:

```
C***Initialize the direction.
    FBR = 'Forward'
C***Search the entire Symbol Table.
100 CALL IE(Symbol Table, FLAG)
```

(continued on next page...)

(continued)

```
C***Reset the direction.
    FBR = 'Forward'
C***Determine if the entire Symbol Table has been traversed.
    IF (FLAG indicates end-of-list) GO TO FINISHED
C***The Symbol Table has not been completely traversed.
C***Determine if this element in the Symbol Table is a scalar.
    CALL GET(CLASS)
    IF (CLASS.EQ.SCALAR) GO TO 200
C***The Symbol Table entry is not a scalar.  Continue traversing the list.
C***The flow of control is about to go backward.
    FBR = 'Backward'
    GO TO 100
C***The Symbol Table entry is a scalar.  Now, investigate how it is used.
C***Enter the Linked List Usage Table from the Symbol Table.
200 CALL TT(Linked List Usage Table, FLAG)
C***Reset the Direction.
    FBR = 'Forward'
C***Determine if the entire list has been traversed.
    IF (FLAG does not indicate end-of-list) GO TO 210
C***The list has been completely traversed.
C***The flow of control is about to go backward.
    FBR = 'backward'
    GO TO 100
```

(Continued)

```
C***The list has not been completely traversed.  Determine if the Usage
C***Table entry is used as an input to an assignment statement.
210 CALL GET(USAGE)
    IF (USAGE.EQ.Input-to-an-assignment-statement) CALL OUTPUT
C***Continue traversing the list.  The direction is backward.
    FBR = 'Backward'
    GO TO 200
        .
        .
        .
```

Approximate Air Source Code

```
C***EOL MEANS END-OF-LIST
    FBR = HF
100 CALL IE(HSYM,BFLAG)
    FBR = HF
    IF(BFLAG.EQ.EOL) GO TO 1000
    CALL GETE(HSYM)
    IF (CLASS.EQ.SCALAR) GO TO 200
    FBR = HB
    GO TO 100
```

(continued)

```
200 CALL TT(HUSE1,BFLAG)
    FBR = HB
    IF(BFLAG.NE.EOL) GO TO 210
    FBR = HB
    GO TO 100
210 CALL GETE(HUSE1)
    IF (USAGE.EQ.(input-to-assignment-statement)) CALL OUTPUT
    FBR = HB
    GO TO 200
```

Initially, the Control Stack is empty.

The first time the Initial Entry subroutine is executed, the Control Stack contains

Control Stack

Module Number	Table Name	List Indicator	Pointer
3	SYM	699	1

where the List Indicator contains the number of entries in the list still unexamined and the Pointer points to a row in the Symbol Table (the first element in the list).

Row 1 of the Symbol Table is empty, therefore, it cannot have a class of 'scalar'. Flow of control branches back to the IE subroutine with FBR set to 'backward'.

The Control Stack now contains

Control Stack

3	SYM	698	2

Row 2 of the Symbol is empty, and the flow of control again branches back to the Initial Entry Subroutine with FBR set to backward. This sequence continues. When row 12 of the Symbol Table is reached, the Control Stack contains

Control Stack

3	SYM	688	12

Row 12 of the Symbol Table does not contain a scalar. The flow of control remains unchanged until row 322 of the Symbol Table is reached. The Control Stack now contains

Control Stack

3	SYM	378	322

The entry in row 322 in the Symbol Table is a scalar. The flow of

control now branches to the statement containing a call to the Table to Table Transition subroutine, with FBR previously set to forward. After TT has been executed, the Control Stack contains

Control Stack

3	USE1	1	2
3	SYM	378	322

where the List Indicator indicates that the end of the linked list in the Usage Table has not been reached, and the Pointer points to the second row in the Linked List Usage Table (USE1).

The entry in the Usage Table represents a usage of 'an input variable in an assignment statement'. This represents a complete template match, and an appropriate message is printed. The flow of control now branches back to the call to the TT subroutine, with the Forward-Backward Register set to backward.

After the TT subroutine has been executed, the Control Stack contains

Control Stack

3	USE1	0	4
3	SYM	378	322

The Forward-Backward Register is set to 'backward' and the flow of control branches back to the IE subroutine. After IE is executed, the Control Stack contains

Control Stack

3	SYM	377	323

Processing continues as before until the last row (last element in the list) is being examined. The Control Stack contains

Control Stack

3	SYM	0	700

Row 700 is empty. IE is again executed. This time, it is discovered that the List Indicator indicates that the list in the Symbol Table has been completely traversed. The Control Stack is popped and the FLAG is set to indicate that there are no more list entries to examine. The search has been completed.

Traversing Lists

As discussed in the section "AIR Basic Search Technique", lists are traversed through the Initial Entry subroutine (IE) and the Table to Table Transition subroutine (TT), with reference to the Forward-Backward Register (FBR).

That discussion can be summed up as the following:

1) IE can only reference a list that consists of an entire table.
   a) If IE is called with FBR set to 'forward', than a table is entered at its beginning, i.e., the search examines the first row of the table.
   b) If IE is called with FBR set to 'backward', then the search examines the next row in the table.
2) TT can only reference a list which can be entered from another list.
   a) If TT is called with FBR set to 'forward', then a list is entered from another list, and the entry point row is examined by the search.
   b) If TT is called with FBR set to 'backward', then the next row in the list is examined by the search.

Although the lists of IE and the lists of TT occupy the same physical space in the tables, the lists of one cannot be referenced by the other. The lists of IE and those of TT are mutually exclusive.

IE can only reference certain tables, i.e., lists. These tables are the following:

1. COM (COMMON Block Name Table)
2. DIR (Directory)
3. IS (Inverse System Hierarchy Table)
4. NOD (Node Table)
5. SH (System Hierarchy Table)
6. SYM (Symbol Table)
7. USE1 (Linked List Usage Table)
8. USE2 (Statement Number Linked Usage Table)

The lengths of the above lists is equal to the contents of the pointer to the last non-empty row in the table. The only exception is the Symbol Table. Because it is a hash-coded table, the length of its list is equal to the physical length of the table.

Whenever IE is called and FBR indicates'forward', the List Indicator in the Control Stack is set to the length of the list minus one. Afterwards, when IE is called and FBR equals 'backward', the List Indicator is decremented by one. When the List Indicator equals zero, then the end of the list has been reached.

When entering a list at its beginning, TT can only reference lists which can be reached through other lists. These lists reside in separate tables; a transition from one table to another is necessary.

The permissible paths from Table 1 to Table 2 are as follows:

Table 1		Table 2	Method
COM	→	LIN	pointer to Linked List Table
DIR	→	SYM	hash and search
DIR	→	SH	row index
DIR	→	IS	row index
IS	→	ISD	pointer to Inverse System Hierarchy to Directory Table
ISD	→	DIR	pointer to Directory
LIN	→	DIR	pointer to Directory
NOD	→	USE2	USETAB Pointer
NOD	→	SUC	Successor Pointer
NOD	→	PRE	Predecessor Pointer
PRE	→	NOD	statement number
SH	→	SHD	pointer to System Hierarchy to Directory Table
SHD	→	DIR	pointer to Directory
SUC	→	NOD	statement number
SYM	→	USE1	USETAB Top Pointer
SYM	→	DIR	sequential search
USE1	→	SYM	use Back Pointer and Back Pointer Code to find beginning of linked list; then use Back Pointer to SYMTAB

(These Tables (1 and 2) are continued on the following page).

Table 1		Table 2	Method
USE1	→	USE2	find first occurrance of the associated statement number
USE1	→	NOD	statement number
USE2	→	USE1	use Back Pointer and Back Pointer Code to find beginning of linked list
USE2	→	NOD	statement number

When TT is called and FBR indicates 'forward', the List Indicator in the Control Stack is initially set to one, except when the list is empty, in which case it is set to zero, or except for the following cases:

TAB1		TAB2	
NOD	→	SUC	List Ind. = Successor Number Column Entry
NOD	→	PRE	List Ind. = Predecessor Number Column Entry
IS	→	ISD	List Ind. = Number of Entries Column Entry
SH	→	SHD	List Ind. = Number of Entries Column Entry

In these four transitions, from TAB1 to TAB2, the List Indicator is set to the length of the list in TAB2.

When TT is called and FBR indicates 'backward', TT determines if the end of the list has been reached.

For all but the lists enumerated below, the length of the list is known. If the list resides in one of the tables discussed above, its length is derived from another table, or else its length is one.

Each time TT is called with FBR set to 'backward', the List Indicator is decremented by one until the List Indicator contains a zero. This means the end of the list has been reached.

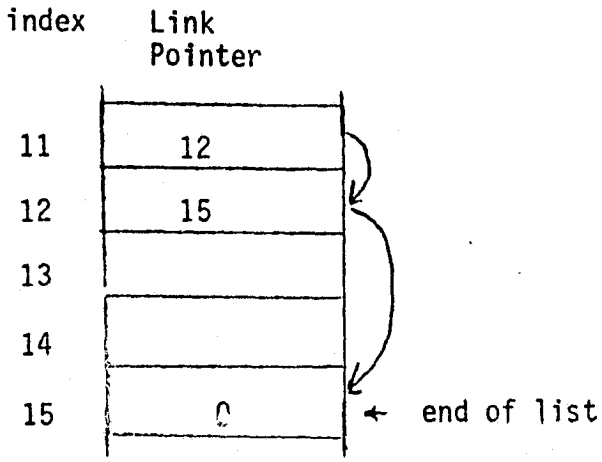
For the following lists, the length of the list is never known. TT determines that the end of the list has been reached when the following conditions hold:

Table	Linkage	End of List Condition
USE1	Linked List	When Forward Pointer = 0
USE2	Sequential	Upon entry to USE2, if statement number = n, When statement number > n
LIN	Linked List	When Linkage Pointer = 0

Example for USE2:

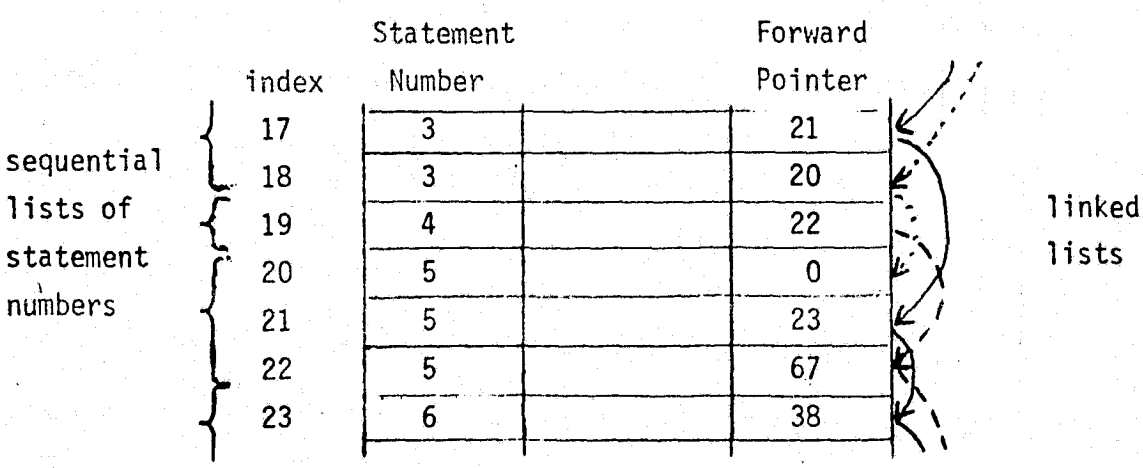
index	statement number	
25	7	
26	8	← Entry point into list, n = 8
27	8	
28	8	← End of list, n = 8
29	9	← Next list, m = 9, m > n

Example for linked list:



Unlike any other table used in AIR, the Use Table contains two types of lists which can be accessed by TT, sequential lists of statement numbers and linked lists. The linked lists connect all references to a specific symbol string in the Use Table. The sequential lists of statement numbers are the sequential appearances of a specific statement number in the Use Table.

Use Table (abbreviated)



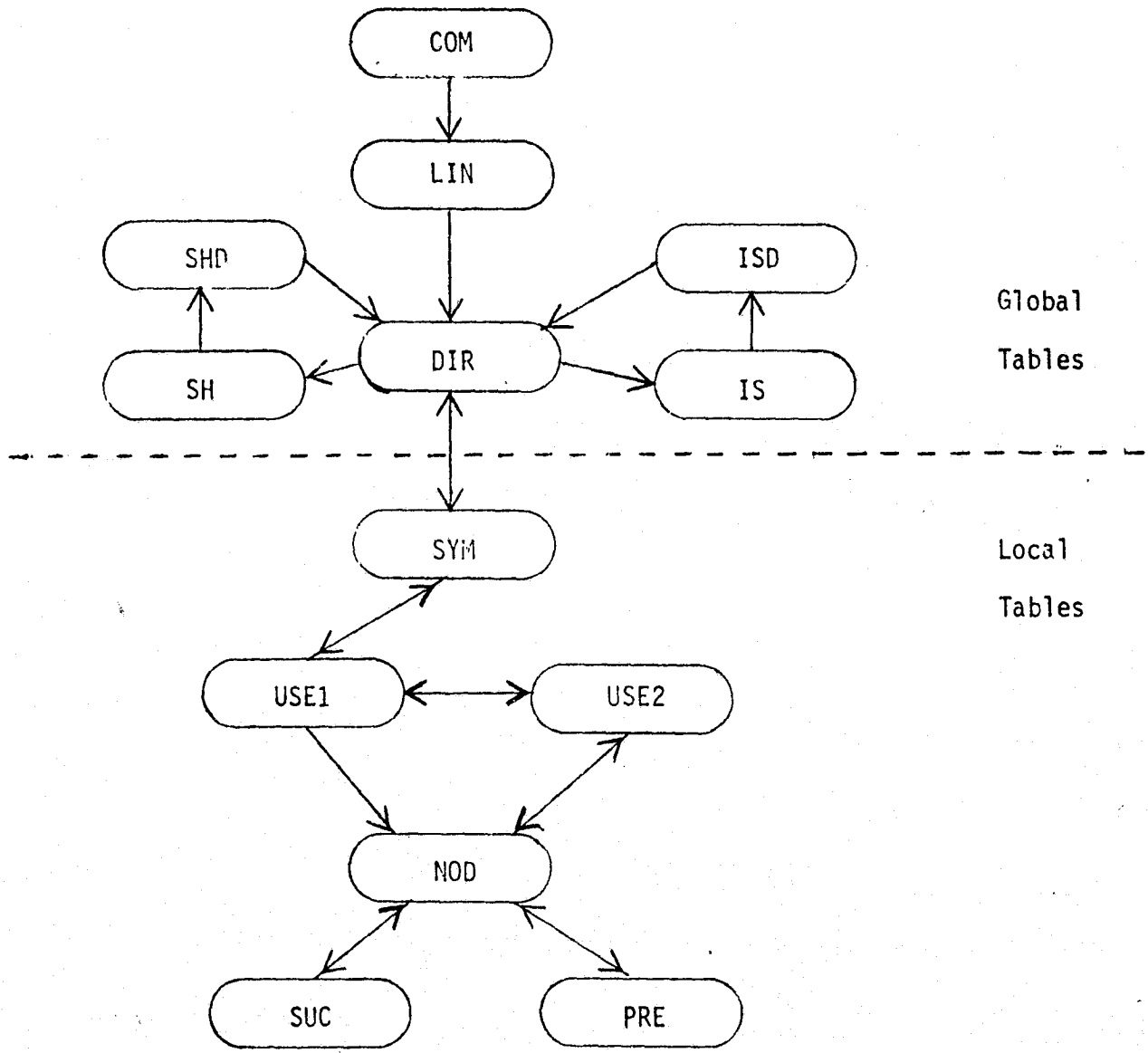
Because TT must have the capability of accessing both lists, AIR was designed to view the Use Table as two logical tables, both occupying the same physical space. The linked lists are in the Linked List Usage Table (USE1), while the sequential lists of statement numbers are in the Statement Number Linked Usage Table (USE2).

Important:

Note that USE1 and USE2 occupy the same physical space. They also use the same length variable (LUSE), the same current row pointer (PUSE), and the same pointer to the last non-empty row (PLUSE).

Designer's Comment: In retrospect, USE1 and USE2 should have their own current row pointers, PUSE1 and PUSE2.

Legal Table to Table Transitions



Calling Sequence Path Tracing

A path is placed in the Trace Stack, one node at a time, starting at a specified beginning point. Path construction continues until one of the following conditions occurs:

a) The node at the top of the Trace Stack has no successors,

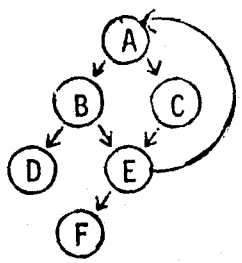
b) The path becomes circular, i.e., the path branches back onto itself.

After one of the above conditions has occurred, the path is backtracked, one node at a time, until either a node is reached that has an as yet unexamined successor or until the path is backtracked to its starting point, i.e., the stack is empty. If a node has unexamined immediate successors, then the trace of the path is again extended. If the Trace Stack is empty, then no more processing occurs for paths beginning at the specified starting point.

The immediate successors of a node are kept in a sequential list in the System Hierarchy Tables. When a node is placed into the Trace Stack, the second column is set to the number of immediate successors in the table, and the third column is set to the first immediate successor in the sequential list. Each time an immediate successor is added to the Trace Stack, the count column (column two) is decremented by one, and the pointer column (column three) is incremented by one. Thus, the count column contains the number of unexamined immediate successors, and the pointer column points to the next successor to be examined. When the count column at the top of the Trace Stack contains a zero, then the node at the top of the Trace Stack has no unexamined immediate successors.

The following example illustrates the above discussion.

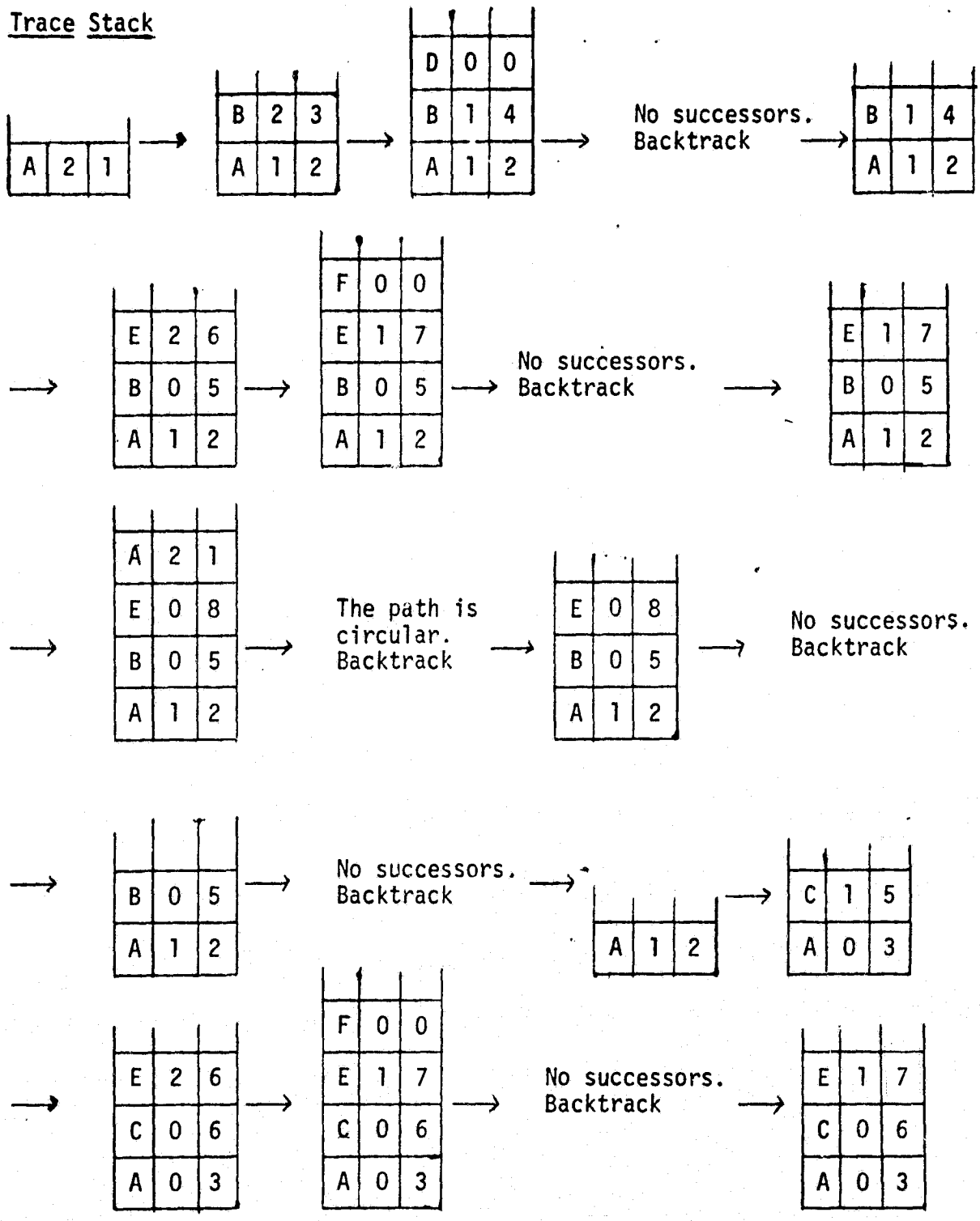
Node Diagram

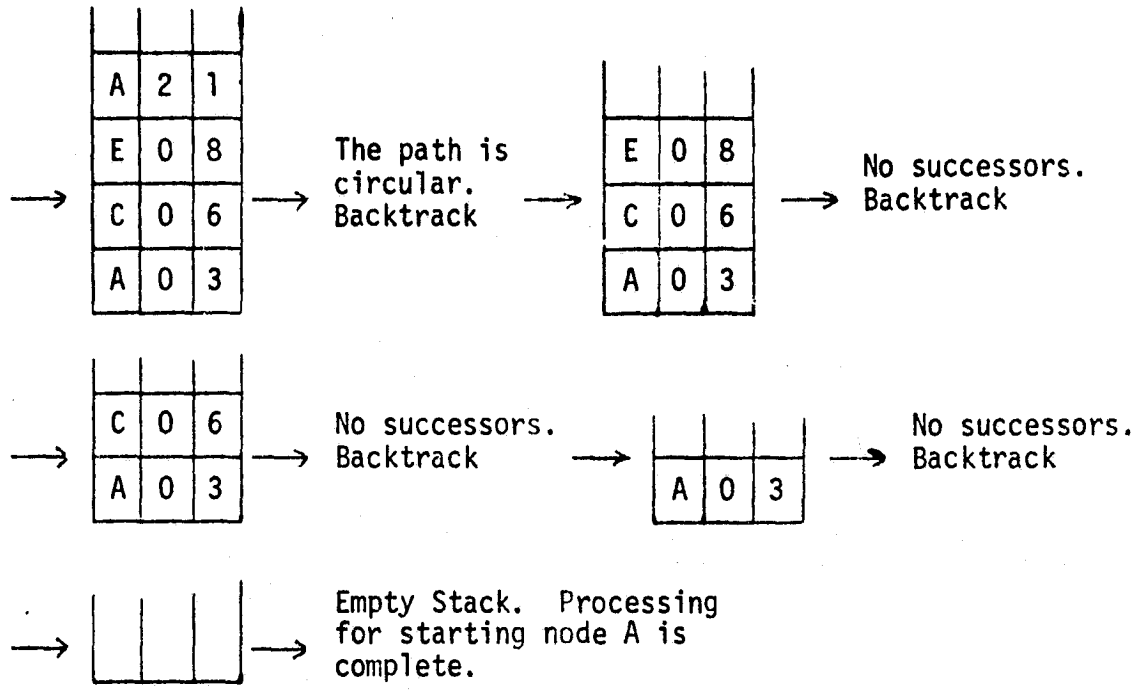


Tables

Node	Number of Successors	Pointer to First Successor
A	2	1
B	2	3
C	1	5
D	0	0
E	2	6
F	0	0

	Sequential Successor List
1	B
2	C
3	D
4	E
5	E
6	F
7	A


Trace Stack
No successors. Backtrack
No successors. Backtrack
The path is circular. Backtrack
No successors. Backtrack
No successors. Backtrack
No successors. Backtrack


A 2 1
E 0 8
C 0 6
A 0 3
The path is circular. Backtrack
E 0 8
C 0 6
A 0 3
No successors. Backtrack
C 0 6
A 0 3
No successors. Backtrack
A 0 3
No successors. Backtrack
Empty Stack. Processing for starting node A is complete.

Flow of Control Path Tracing

A path is placed into the Path Stack, one node at a time, starting at a pre-specified beginning point. The construction continues until one of the following conditions occurs:

a) the node at the top of the Path Stack has no successors,

b) the path has reached the end node (the pre-specified ending point),

c) the path has become circular (the path branches back onto itself).

After a path has been built, a backtrack operation occurs. This means that the node at the top of the Path Stack is removed, and the next node on the Path Stack is checked to see if it has any alternate successors (alternate edges). If there are alternate successors, the first successor is chosen, and a new path is built which passes through the successor node. If there are no successors, the backtrack operation continues until the path has been backtracked to its starting node. This means that all non-circular paths, from a beginning point to an end point, have been investigated.

All of the successors of a node are placed in the Alternate Edge Stack. As each successor is needed, it is removed from the Alternate Edge Stack and placed on the Path Stack.

The Path Stack consists of one column. Each entry is a node in the same path. The bottom of the Path Stack contains the beginning node of the path, and the top contains the node furthest down the path that the system has yet reached.

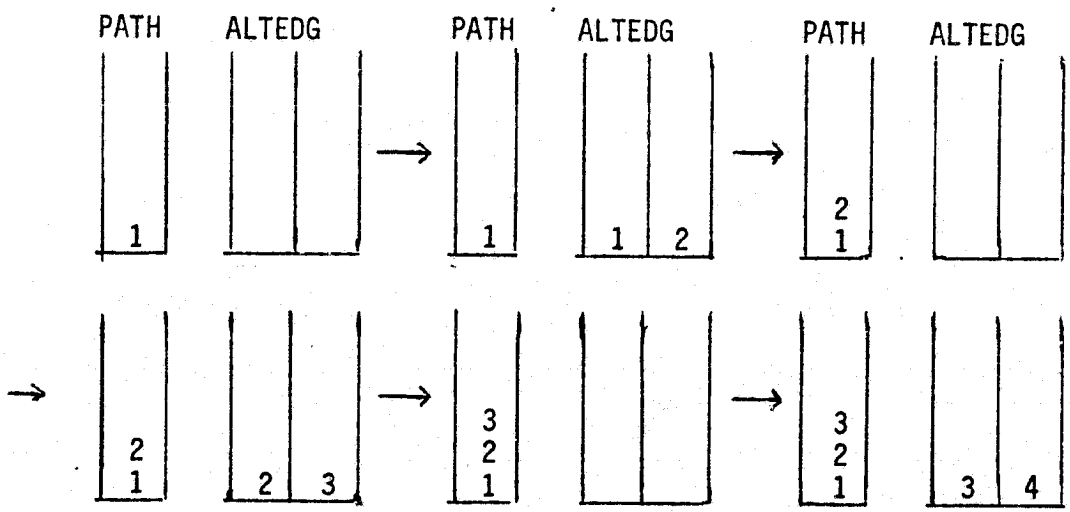
The Alternate Edge Stack consists of two columns. The first column contains a node that exists in the Path Stack. The second column contains a successor to that node.

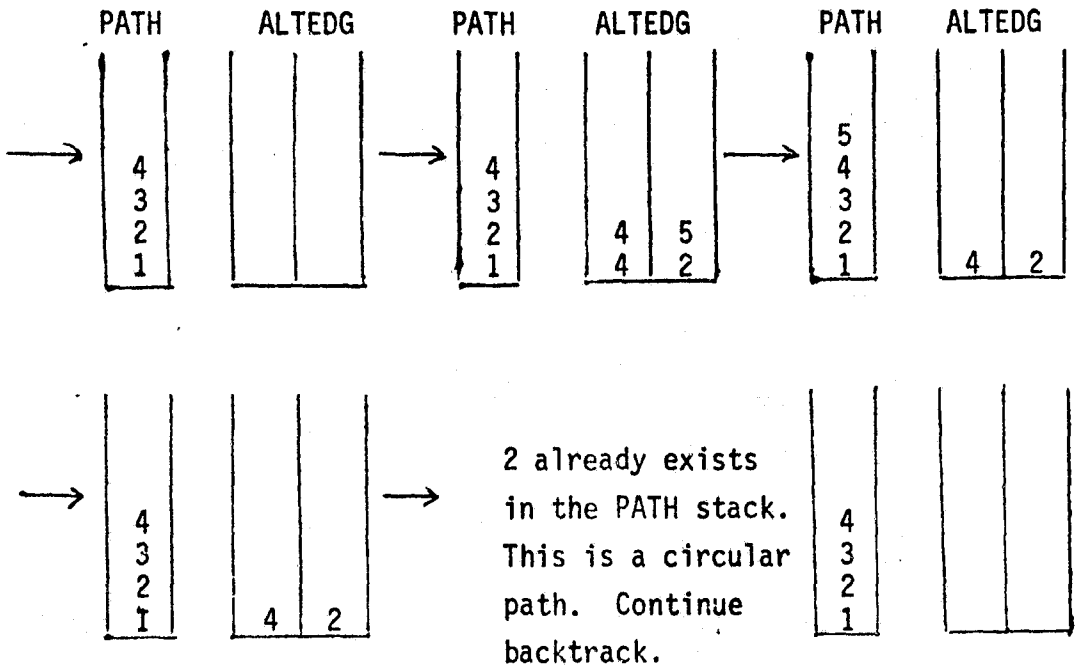
A node may have no alternate successors in the Alternate Edge Stack, in which case it will not appear in that stack. On the other hand, multiple successors means multiple occurrences of a node in the Alternate Edge Stack.

The following examples illustrate the above discussion.

Example 1)

```
1  A = B                    Beginning Point = 1
2  DO  4 I = 1, K           Ending Point = 5
3      C = D
4  CONTINUE
5  RETURN
```





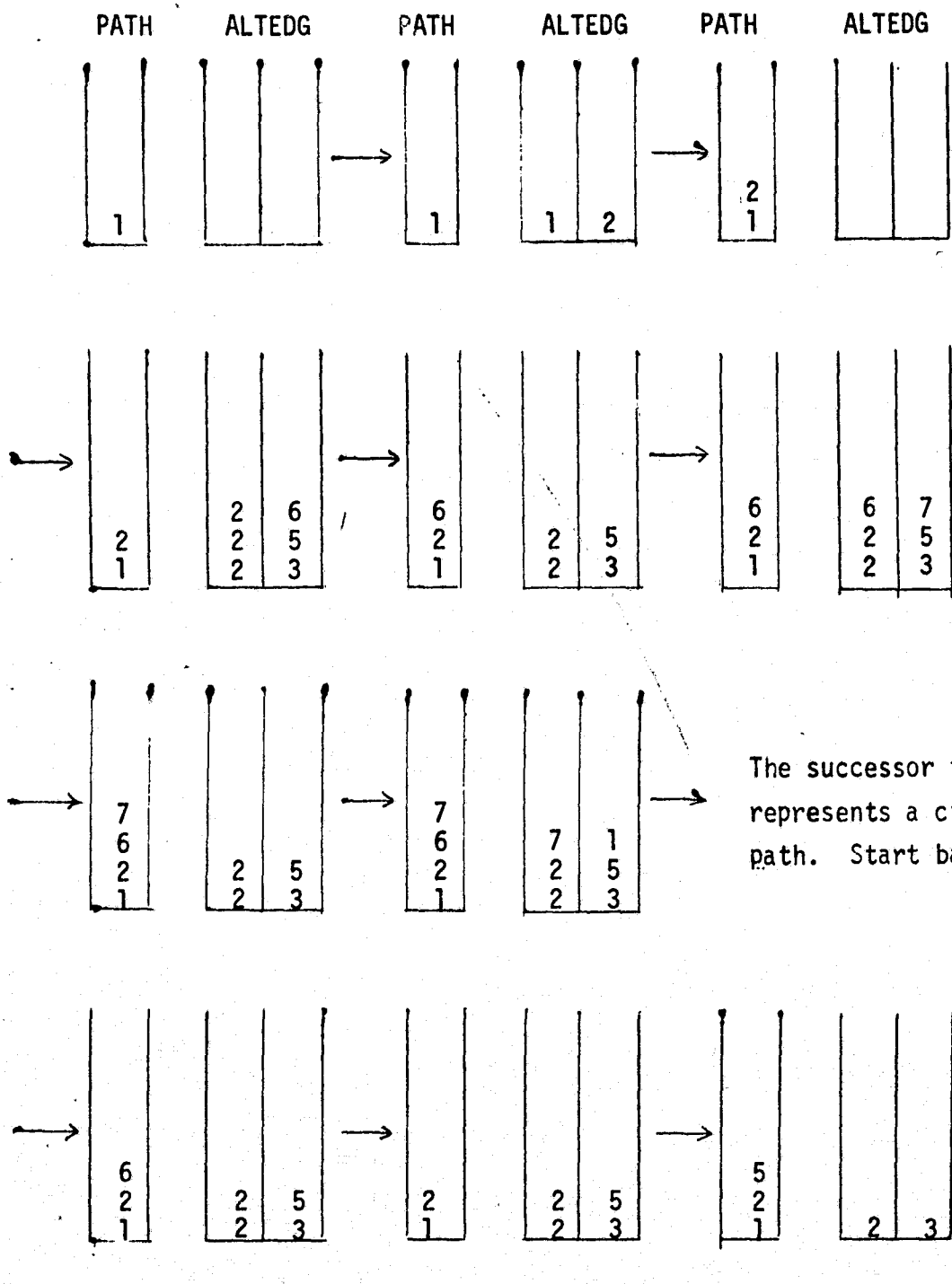
At this point, the Alternate Edge Stack is empty. This implies that there are no more possible paths to traverse. Thus, the path search through all possible paths has been completed. End of computation.

Example 2)

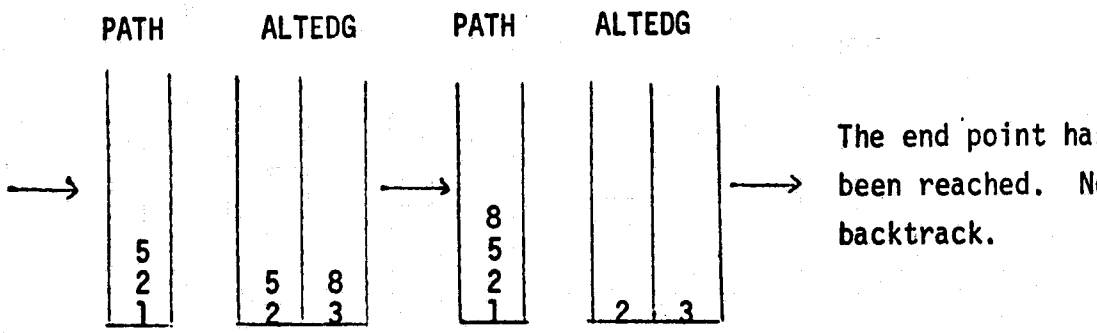
```
1   K = Z + J
2   GO TO (3, 5, 6), K
3   J = M
4   GO TO 1
5   GO TO 8
6   J = L
7   GO TO 1
8   RETURN
```

Beginning Point = 1

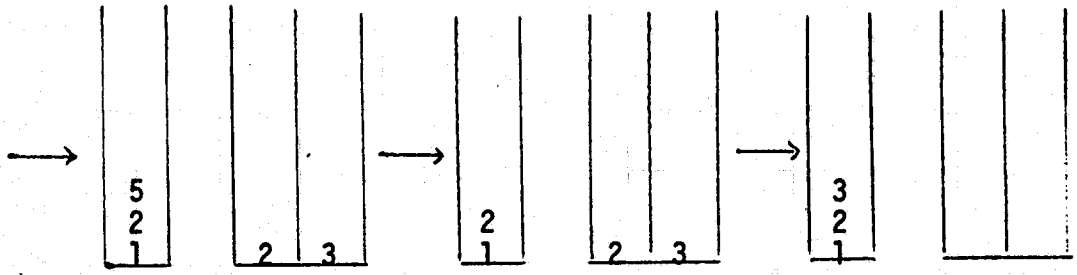
Ending Point = 8

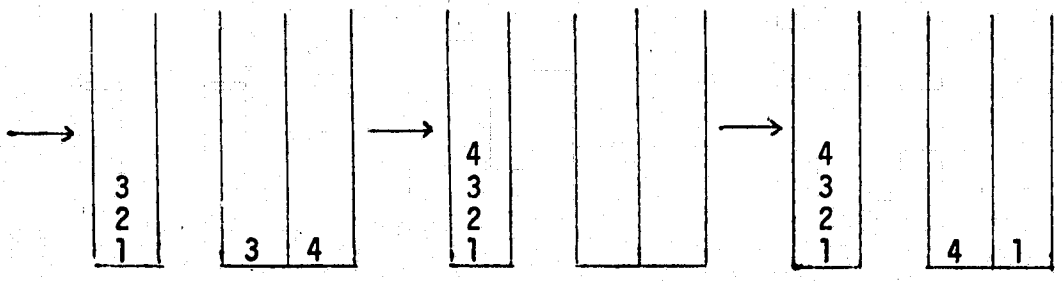


The successor to 7 represents a circular path. Start backtracking.

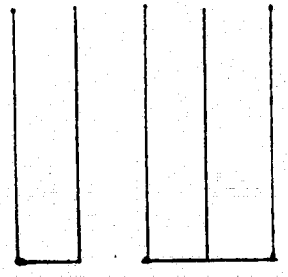


The end point has been reached. Now backtrack.





The only successor of 4 is already in the Path Stack. Backtrack.



The Alternate Edge Stack is empty in the backtrack mode. This implies that all non-circular paths between the beginning point and end point have been traversed. End of computations.

Rudimentary Discussion of Phase 2

AIR - Driver of the AIR subsystem.

ALPHA - Determines if a character is an alphanumeric character.

ANSIST - Search for ANSI Standards function names not used as ANSI Standards functions.

ASNUSE - Search for local variables assigned values but never used.

ATFILE - Attach a list in a file to the List Table.

BLDCIT - Builds command item from command card data.

CBDIM - COMMON Block Alignment Check for entry dimensionality mismatch.

CBINDS - COMMON Block Alignment Check for individual entry size mismatch.

CBNAME - COMMON Block Alignment Check for entry name mismatch.

CBNENT - COMMON Block Alignment Check for total number of entries mismatch.

CBTOTS - COMMON Block Alignment Check for total size mismatch.

CBTYPE - COMMON Block Alignment Check for entry type mismatch.

CMDEND - Terminal command card procedure.

COMBAL - Driver of the COMMON Block Alignment Check.

CONALC - Construct the Alignment Tables for the COMMON Block Alignment Check.

CONALP - Construct the Alignment Tables for the Parameter List Alignment Check.

CONCOM - Construct the COMMON Block Reference Tables.

CONISH - Construct the Inverse System Hierarchy (the Called-by Hierarchy) Tables.

CONSH - Construct the System Hierarchy (the Calling Hierarchy) Tables.

CONVER - Convert a vector of decimal digits in A1 FORMAT into an integer value.

CONVRT - Convert an alphanumeric character string into its integer value.

CYCALL - Search for cyclic calling sequences.

DATVAR - Search for DATA statements not in BLOCK DATA which contain COMMON Block variables.

DEL - Delete a list and all lists that follow it from the List Table.

DIGIT - Determine if a character is a decimal digit.

DMPAIR - Dump the AIR control structures.

DMPCOM - Dump the COMMON Block Reference Tables.

DMPSHT - Dump the System Hierarchy Tables and Inverse System Hierarchy Tables.

DOTERM - Search for DO Loop index variables used after the DO Loop terminated normally.

EQUIVL - Place the name pointed to by the top of the Control Stack and any names EQUIVALENCED to it into a list in the List Table.

ERHALT - An internal terminal error has occurred. Halt AIR processing.

FACES2 - Driver of Phase 2.

FELCOM - Fetch an element from the COMMON Block Name Table.

FELDIR - Fetch an element from the Directory.

FELIS - Fetch an element from the Inverse System Hierarchy Table.

FELISD - Fetch an element from the Inverse System Hierarchy to Directory Table.

FELLIN - Fetch an element from the Linked List Table.

FELNOD - Fetch an element from the Node Table.

FELPRE - Fetch an element from the Predecessor Table.

FELSH - Fetch an element from the System Hierarchy Table.

FELSHD - Fetch an element from the System Hierarchy to Directory Table.

FELSUC - Fetch an element from the Successor Table.

FELSYM - Fetch an element from the Symbol Table.

FELUSE - Fetch an element from the Use Table.

FILCLS - Marks and closes Phase 2 files.

FILOPN - Opens and positions Phase 2 files.

FLD - Manipulate bits of a word as would UNIVAC's FLD function.

FUNPAR - Search for function dummy parameters which are assigned values within the function itself.

GETE - Get an element from a table.

GETL - Get the local tables of a module (bring them into main memory).

GETSCA - Get a scalar.

HASHSY - Hash into the Symbol Table.

IE - Perform a initial entry into a table. Also follows lists.

IMPLDO - Determine if a variable is set by an implied DO Loop.

INCOM - Bring the COMMON Block Reference Tables into main memory.

INDIR - Bring the Directory into main memory.

INGHD - Bring the Global Header into main memory.

INISH - Bring the Inverse System Hierarchy Tables into main memory.

INSH - Bring the System Hierarchy Tables into main memory.

INTAIR - Initialize the AIR subsystem.

LIRL - Load the Immediate Register from a list in the List Table.

LNKAIR - Processes user query descriptions, initiates AIR process, and initiates and terminates files for AIR processing.

MANL - Manipulate a list's description in the List Table Map.

MODNAM - Find the name of a module.

MULBRA - Search for multiple branching statements which do not branch to the statement immediately following.

NXTCMD - Acquires next command item from the command card.

OUTCOM - Move the COMMON Block Reference Tables out to secondary storage.

OUTGHD - Move the Global Header out to secondary storage.

OUTISH - Move the Inverse System Hierarchy Tables out to secondary storage.

OUTSH - Move the System Hierarchy Tables out to secondary storage.

PARAL - Driver of the Parameter List Alignment Check.

PATHS - Build paths which reflect the flow of control through a module.

PLDIM - Parameter List Alignment Check for parameter dimensionality mismatch.

PLNENT - Parameter List Alignment Check for total number of parameters mismatch.

PLTYPE - Parameter List Alignment Check for parameter type mismatch.

POP - Pop the Control Stack.

PRTQLS - Prints queries being performed by AIR

PUSH - Push the Control Stack

RDCTRL - Read of command card.

READLT - Read in the local tables of a module.

REDLOP - Search for DO Loop control variables assigned values within the DO Loop itself.

RESWRD - Search for FORTRAN 'reserved' words used as names.

SETSCA - Set a scalar.

SRCHDI - Search the Directory for a name.

TCOM - Perform a table to table transition from the COMMON Block Name Table.

TDIR - Perform a table to table transition from the Directory.

TIS - Perform a table to table transition from the Inverse System Hierarchy Table.

TISD - Perform a table to table transition from the Inverse System Hierarchy to Directory Table.

TLIN - Perform a table to table transition from the Linked List Table.

TNOD - Perform a table to table transition from the Node Table.

TPRE - Perform a table to table transition from the Predecessor Table.

TRACHI - Trace the calling hierarchy.

TSH - Perform a table to table transition from the System Hierarchy Table.

TSHD - Perform a table to table transition from the System Hierarchy to Directory Table.

TSUC - Perform a table to table transition from the Successor Table.

TSYM - Perform a table to table transition from the Symbol Table.

TT - Driver of table to table transitions. Also follows lists.

TUSE1 - Perform a table to table transition from the Linked List Use Table (USE1).

TUSE2 - Perform a table to table transition from the Statement Number Linked Use Table (USE2).

UNINT - Search for uninitialized local variables.

USERQ - Interprets user requests from command card and constructs query list to be performed.

Construction Conventions of AIR

I. Malfunctions

There are two classes of malfunctions that can occur in AIR.

1. Bad data in the tables produced by the FFE. AIR will try to bypass any bad data. No error messages are printed.
2. AIR has committed an internal processing error. How AIR reacts to the malfunction is determined by where the malfunction was detected.
   a. The malfunction was detected by a query. An error message is printed. Processing on the query halts, although system processing continues.
   b. The malfunction was detected by a utility. An error message is printed. No recovery is attempted. AIR processing halts.

II. FORMAT Statements

1. A FORMAT statement immediately follows the first I/O statement which references it.
2. FORMAT labels are always odd numbers. The last digit in the FORMAT label is almost always '1'.

III. Statement Labels

1. Statement labels ending with '00' or '000' delineate major sections of code.
2. Transfer labels are always even numbers.
3. FORMAT labels are always odd numbers.
4. Sections dealing with internal AIR errors have statement labels in the 900's and 9000's.

5. All statement labels are either used in transfers or are used as FORMAT labels.
6. All statement labels are in increasing order.

IV. DO Loops

1. DO Looops are intended.
2. Each DO Loop ends with its own CONTINUE statement.

V. Non-executable Statements.

1. Non-executable statements, except for FORMAT statements, precede all executable statements.
2. COMMON Block declarations appear in alphabetical order.

VI. Protection Code

Code which es executed only if bad data is introduced during AIR processing is immediately preceded by

C****PROTECTION CODE

and immediately followed by

C****

VII. REPORT GENERATOR

Design Considerations

Purpose. The Report Generator subsystem produces user displays of information extracted by analysis of the software system.

Requirements. The Report Generator produces the primary interface with the user--report results. The report form should be easy to follow. Report information should clearly identify how the analysis results were detected. Information displayed should completely describe, yet not bury, the user in printout.

The Report Generator should not process FORTRAN constructions. Interpretation of FORTRAN should be supplied by other system components.

The Report Generator should be isolated in so far as possible from numerical codes assigned by other subsystems. Relational and structural control tests are preferred to numerical value tests.

Need for reference manuals and auxiliary listings should be minimized in interpreting the reports produced. Reference materials should be required only for suspected malfunctions and subtle problems.

The Report Generator should shield the user from redundant and superfluous information generated by other subsystems.

Reports should contain connected source code events even if the lines referenced are contained in different modules. Related problems should be displayed on a single listing.

Strategy. Users will be most familiar with compiler listing forms of program displays. Therefore, a listing is used for display results. This format provides not only a familiar form but also permits other

lines of interacting code to be examined.

To isolate the Report Generator from message codes, structural relationships (e.g., A .GT. B) are used between present/last values and current/next values of data items. Only the lowest level routines use numerical values extensively.

To minimize I/O time on source code extraction, report generation follows the order of the Source Code Catalogue.

For generality in operation, the Report Generator is largely driven by the contents of the analysis Flag file entries. Flag data values cause lines of source code to be extracted for display. Boundary conditions detected in report messages control listing boundaries. Listing boundaries cause printout positioning and report heading prints.

The Flag sorting is exploited to achieve the following results:

1. Source code reference order of analysis messages is the same as the Source Code Catalogue entries.
2. Messages generated by independent routines are ordered on the same line of source code text. Flags associated with a single line group of source code are adjacent on the Flag file.
3. Redundant Flag information becomes adjacent on the Flag file.

Overview of Report Generator Operation

Types of Reports

1. Primary Reports. Primary reports are full module listings annotated with analysis messages.

2. Secondary Reports. Secondary reports are composed of data drawn from analysis investigations and selective lines of code extracted from modules. Classifications of secondary reports are:

   a. Display Reports. Displays of data extracted by program analysis for which source code is not needed.

   b. Secondary Listing Reports. Truncated listings of one or more modules generated to show a fragment of program operation or provide adjacent display of source lines from several modules.

Initial Conditions.

When the Report Generator is activated, source code for the modules examined resides on the Source Code Catalogue File (SCAT) and sorted flags from AIR and FFE reside on the Flag File (FLAG). The sort order produces a sequence of flags associated first by global key number and secondly by order of the source code on the Source Code Catalogue.

The primary report is always generated first, followed by any secondary reports needed. Either report set may be empty (i.e., no data printed for the report). Control of the report generation process is determined by the report options selected and data contents of the Flag File.

For report production, individual Flag File entries are combined together into units called "messages". Messages are portions of analysis results tied to a given set of source code lines with data attached for a particular analysis result. For example, Flags generated by CBTYPE which are associated with one line of code, are collected together as a message.

Report Production

Report Production exploits sorted aspects of the Flag File contents and Source Code Catalogue order. Report generation control centers on the options selected in the REPORT card and value contents of the Flag File entries. The following events control report production:

1. Selection of the ALL option causes modules to be selected from the directory entries in the order they appear on the Source Code Catalogue.
2. The FLAG option causes modules to be selected in the order indicated by the sorted contents of Flag File entries.
3. A change in the source code origin of Flag entries indicates a new module's source code is being referenced.
4. Global sort key values are 1 for primary messages and +N for secondary report messages.
5. The end of a report is identified by a change in the global key. Each report has a unique key value.

The primary report is always produced first. The primary report consists of a series of annotated source code listings followed by a

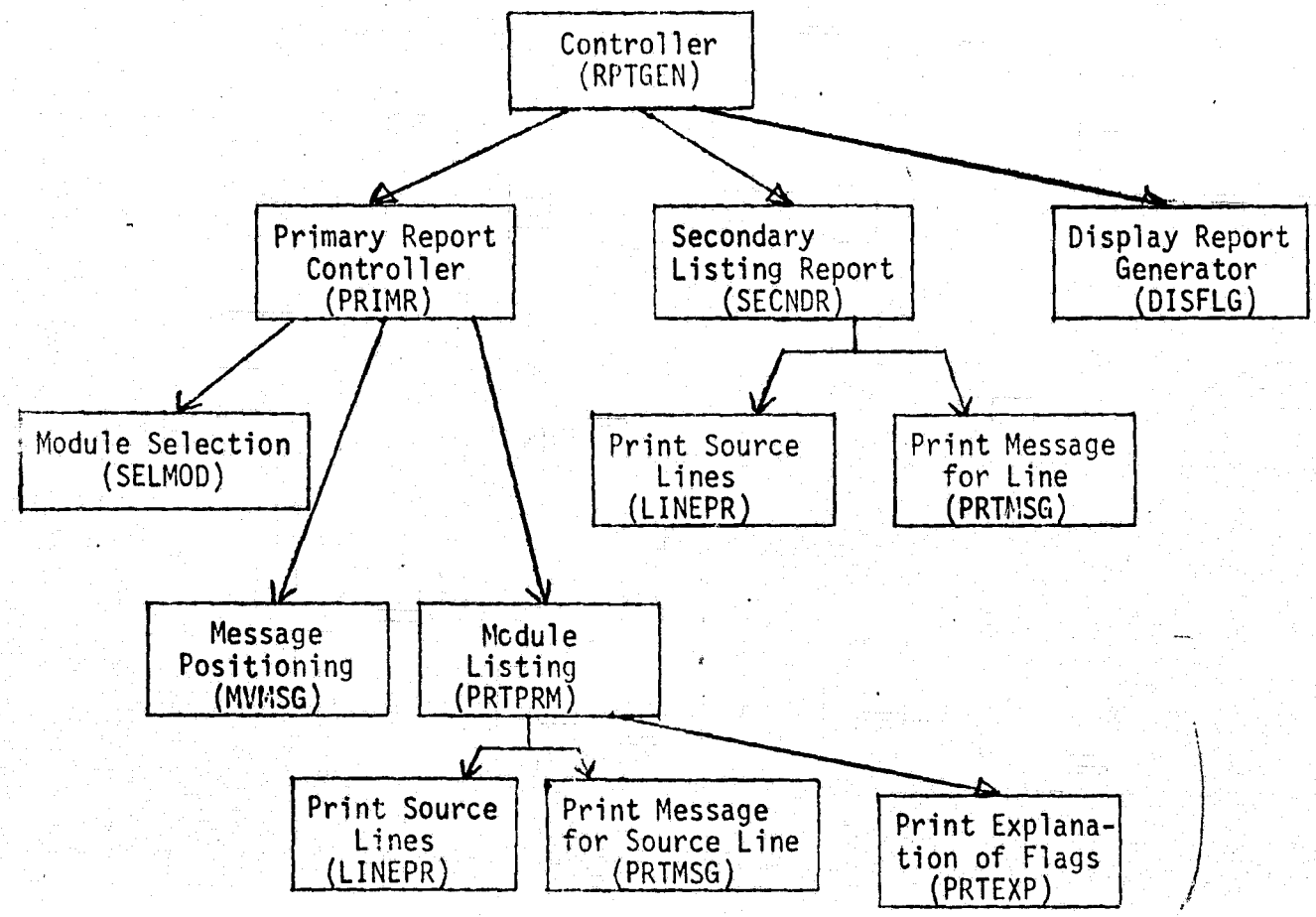
legend of explanations for the generated messages. The order of the listings is the same as the order of the card images on the Source Code Catalogue. Since the flags are sorted in the same order as the source code, flags appear in an increasing sequence of card images.

Primary Report Production. The primary report production requires the following actions:

1. Selection of the next module to print.
2. Control listing of the modules and insertion of annotations for messages.
3. Provide a descriptive legend of explanations for the annotations.

Figure VII-1

Report Generator Routine Hierarchy



Selecting the next module is governed by the user option and the last module processed. If the ALL option is selected, all modules are printed regardless of message content. If the FLAG option is selected, only modules containing flags are printed.

In primary reports, the module text is printed, recording the messages generated for the module. At the end of the module, a legend of messages generated for the module is printed.

To print the listing, source lines are printed through the end of the last card indicated by the next message, then control is passed to a message print routine to construct a series of text lines for user display of the message text. If multiple messages appear on the same line, new source text is suspended until all messages for the current lines have been processed. After processing all messages for the selected module, the remaining source text, if any, is printed and control passes to the explanation printing routine.

The next module is selected and the process repeated until all selected modules have been processed.

Secondary Reports. After generating the primary report, a series of secondary reports are issued until secondary messages are exhausted. Secondary messages are those flags having a global key number greater than 1. The principle discrimination between types of secondary reports are whether source code is required for the report.

Each global key change causes a new secondary report to begin. If the first message of the report indicates source code is not required for the report (empty source code origin and first card

descriptors), then the display report processor is activated.

The display report processor produces printed reports using only the data from the flag data fields. Control is returned to the report generator when the data for the report is exhausted (i.e., the key field changes).

Secondary Listing Reports. Secondary listing reports are truncated listings of source code extracted from a portion of one module or from several modules. The messages for a single secondary listing are identified by having the same global key value.

Secondary listings are produced by printing lines of source code indicated by the message followed by the message text itself. If a change of modules occurs during a secondary listing report, space is provided between the source lines of the different modules.

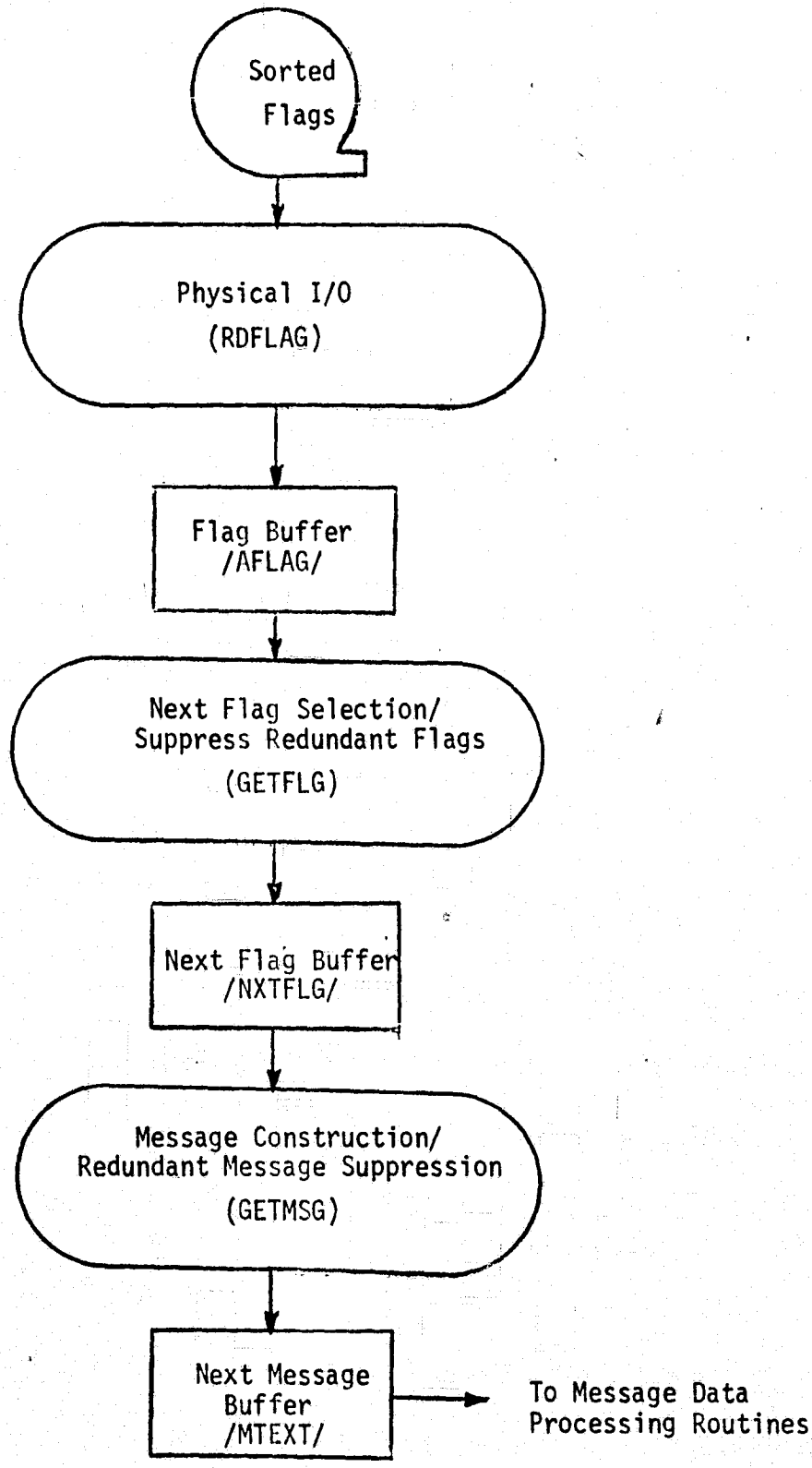
Constructing Messages

Information is passed from analysis subsystems to the Report Generator via Flag File entries. The Flag File contains informational entries to be attached to source code or to be displayed by itself. The Report Generator does not distinguish the source of the information -- only the contents and rules of message construction are used by the Report Generator.

To facilitate report production, the sequential Flag File entries are combined into processing units called "messages". Each message is a logical collection of associated data. An overview of data flow is illustrated in Figure VII-2.

Message Construction Data Flow

Figure VII-2



Combining Flags into Messages. The sort fields of the Flag file entries are arranged to cause all messages associated with a single set of source code to become adjacent on the sorted file. As a secondary effect, the more critical information becomes the first flag for a particular statement.

A series of flags are consumed to construct a message. The first flag of the series establishes the global key number, source code statements for which the message is intended, flag identification information, occurrance number, and internal order number for the entire message. Data text for the first flag becomes the first entry of the message text. Subsequent flags are combined in the same message if they have the same global key number, flag identification, source code pointers, occurrance number and increasing values of internal order. The message is completed when a flag is discovered that does not belong to the current message.

An example of message construction is illustrated in Figure VII-3. The first message is composed of two flags to be attached to the module with source code origin 0, relative card number 6. The second message consists of a single flag to be attached to module source code with origin 34, card number 19.

Notice that messages 5 and 6 are both attached to module origin 34, card 51. In the sort process, the order of the two messages has been reversed; the occurrance number of SABORT is 2, and SYM UREC is 1.

Figure VII-3

KEY	Source Origin	CARD1	CARDN	Flag No.	Flag Symbols	Occurrence	Rel. Order	Integer Data	Alpha Data
1	0	6	6	22	STM TRUC	1	1	0	V V
1	0	6	6	22	STM TRUC	1	2	0	V V
1	34	19	19	12	NO PROC	1	0	0	
1	34	30	30	12	UNPROG	1	0	0	
1	34	40	40	12	NO PROC	1	0	0	
1	34	51	51	23	SAPORT	2	1	0	V (
1	34	51	51	23	SAPORT	2	2	0	V ,
1	34	51	51	82	SYM DREC	1	0	0	,
1	34	59	59	12	UNPROG	1	0	0	
1	34	60	60	23	SAPORT	2	1	0	V (
1	34	60	60	23	SAPORT	2	2	0	V ,
1	34	60	60	82	SYM DREC	1	0	0	,
1	34	66	66	83	NO DEF	1	0	0	85
1	104	18	18	12	NO PROC	1	0	0	
1	104	20	20	12	NO PROC	1	0	0	
1	161	19	19	83	NO DEF	1	0	0	1000
1	201	23	23	53	SYM TRUC	2	1	0	[QUOTE[
1	201	23	23	53	SYM TRUC	2	2	0	S[
1	201	23	23	53	SYM TRUC	2	3	0	
1	201	23	23	53	SYM TRUC	2	4	0	
1	201	23	23	53	SYM TRUC	2	5	0	
1	201	23	23	53	SYM TRUC	2	6	0	
1	201	23	23	53	SYM TRUC	2	7	0	
1	201	23	23	53	SYM TRUC	2	8	0	
1	201	23	23	87	TOC SCON	1	0	0	

I/O and Message Protocalls. Semaphores are used to control I/O and message data transfer. Semaphores detect data not being used and permit nonconsuming examination of data to control sequencing.

When a processor places data in a buffer area, the semaphore is set full (value 1 for single item buffers; length of +N for multiple entry items). The semaphore is set empty (value 0) when the data is consumed (e.g., a message is printed).

Semaphores must be initialized to an empty state for proper operation. Empty indicators stimulate the reading of the first data in the system.

After the data is exhausted, the semaphores assume an empty state causing processing to terminate.

Suppressing Redundant Messages. Flag informtion is assumed to contain potentially redundant entries. This redundant data is suppressed in report generation. Redundancy takes two forms:

1. Multiple adjacent Flag entries.
2. Multiple adjacent messages.

Redundant flags are suppressed by GETFLG. The suppression process requires comparing the next flag to the last flag returned.

Redundant messages are suppressed by GETMSG. A redundant message is identified by comparing the incoming flag sequence to the contents of the last message. Redundant messages may contain superficial differences; redundant messages may not be identical messages. Two messages are redundant if they have:

1. Same key number.
2. Same source code origin and relative card number.
3. Same Flag number.
4. Same occurrance number.
5. New message date is a subset of data provided by the previous message (i.e., no new information).

Redundancy suppression requires the old data values to be maintained intact. Thus, consuming processors of message data must not destroy data values if the redundancy mechanism is to function correctly.

In addition, initial values for data buffers must be established to avoid suppressing the first message or flag as redundant.

Terminal Conditioning. When Flag data is exhausted, data termination must be passed to calling routines. By convention, the end of data is identified by an empty message. To cause termination of the current report, if one is in progress, the key indicator is advanced. For example, if the current key value is 5, an empty message with key value 6 will be generated to cause termination of report 5 via key value and termination of processing by the empty message.

If no flag data is present for a report, an empty message will result with key value 1. This permits the ALL option to print unflagged listing of modules and FLAG option to produce no listings without "special case" flags in the control structures of those routines.

Oversized Messages. Normally, FACES messages are rather short sequences. The size of the message text array is established to accommodate the largest usual length.

Some message sizes are dependent upon the source code characteristics of the software being analyzed. For example, the number of variables equivalenced cannot be predicted. Similarly, the number of modules in a cyclic calling sequence is indeterminate.

Messages longer than the allocated size are called "oversized messages". Acquisition of these messages requires multiple calls to the message construction routine.

On oversized messages, flags are consumed until the message buffer is filled. Remaining flags for the message are withheld until the current message buffer is processed. On the next request for message data, remaining flags are provided until the buffer is filled or the flags are exhausted.

Since the first buffer is lost in the multiple call access, redundant message suppression will not be operative for extremely long messages. Flag producing routines are responsible for avoiding redundant messages where the length of message exceed the buffer size.

Multiple Messages for the same Line. Where several messages occur for the same source line set on a given report, the source lines should be printed only once, followed by the message set. To facilitate this processing, the source line printing routine (LINEPR) contains logic to avoid multiple copies of the same line. By convention, if the first line of the set is numerically greater than the last line of the set, no printing is performed.

The report control routine tracks the last line printed for each module, using the last line as a bound on the next line set. Thus, if

two consecutive cards were found for relative cards 12 through 16, the first message would cause the appropriate lines to be printed. The second call to LINEPR will have "crossed pointers" and no lines will be printed. The message is printed for both cases producing the desired result: a single occurrance of the source code followed by multiple messages.

Message Control. Since message content controls the report sequencing and line printing activities, messages are normally one ahead of current processing. That is, the end of a report is signaled by finding a message for the next report. Under normal conditions, report processors exit with a "live" message in the buffer. Similarly, the end of all reports is indicated by finding an empty message.

The message moving routine (MVMSG) provides control for acquiring the first message to initiate processing. MVMSG also permits recovery if messages and source code lose synchronism through processing error or bad data on the Flag File. Message positioning permits only skipping messages from the incoming Flag File.

Source Code Catalogue Control. The source code catalogue is used in an index/sequential fashion. The source code catalogue is positioned to an initial location, then a series of source lines are extracted for display.

Source code catalogue control is located in the line printing routine (LINEPR). The source code catalogue is usually properly positioned during primary report generation; calls to MVSCAT are redundant protection against lost synchronism. When secondary reports

are being generated, MVSCAT performs the random access to source code from multiple modules or selective access to a subset of module source lines.

VIII. INPUT/OUTPUT FILE DESCRIPTION

FACES uses the simplest file structure possible to enable easy transportation among different host machines. Since resident file manipulation facilities vary widely, preference is given to program resident control of files.

Types of Files. FACES requires two types of files for operation:

1. Sequential files used in standard fashion.
2. Random Access Files used in index sequential fashion.

While operational efficiency will be seriously degraded, analysis could be accomplished using only sequential files.

Sequential Files. Sequential files are utilized for user input, print output, certain processing tables, and intermediate results. Variable length sequential files are terminated by an end of file mark (EOF). Fixed length sequential files are headed by a descriptive record indicating file content and length. Input files to FACES must contain the EOF for proper operation. Sequential files generated by FACES are terminated by an EOF mark during normal operation.

Random Access Files. Random access files are needed for storing analysis tables and program source code presented for investigation. Normally, the random file will be positioned to a given record followed by a series of sequential reads or writes. Where fixed length files are required, the size of the file is carried internally in FACES variables.

Internal file descriptions. Each file is described internally in FACES by a COMMON block dedicated to the file. In addition, file I/O is frequently accomplished through buffers dedicated to I/O; these buffers are also implemented as COMMON blocks. (See System Conventions).

Sequential file manipulations. Sequential files are positioned with the normal rewind and read operations. In addition to simple positioning, sequential files created by FACES may be appended with new data.

To append new data to an existing file, the file is read until an end of file mark is detected. The file is backspaced over the end of file mark, and new data is written on the file. The new data is terminated by a new end of file mark. After writing the EOF, the file is backspaced to permit subsequent read sensing of the file status.

For compatability, empty files are created initially. An empty file is one in which only an EOF mark is present. Due to a system problem encountered during installation, the Flag File is emptied by writing a dummy record on the file followed by an EOF mark. The dummy record is discarded in processing.

I/O Protocalls. Where variable length sequential files (e.g., the Source Code Input File) provide incoming data, I/O protocalls are used to control I/O activities. The protocall involves setting an indicator when physical reads occur and resetting the indicator when the data is used.

The I/O protocall treats several unusual situations unique to I/O.

1. Initial transient. Before the first read, the I/O buffer area contains invalid or empty data. This data should be destroyed by the

first read operation.

2. Nonconsuming inspection. Control may require inspecting the input data to determine correct action. Based upon data contents, the buffer may be used or left for later processing. The indicator permits centralized control of independent routines accessing common data or records the effects of previous calls to the same routine.

3. Final transient. The last read on a variable length sequential file results in an EOF being read. Since this is not normal data, the indicator is used to inform the calling processor that "empty" data was received from the file.

Where the data is acquired into an array buffer, the pointer to the last nonempty entry is used as the I/O indicator. If data is read into fixed scalar variables, a special variable is used for the indicator. The value 0 is used to indicate an empty buffer and a positive integer is used to indicate nonempty contents.

Once an EOF has been read on a file, further calls will not perform I/O activities unless the EOF has been cleared by an external routine. This technique permits error recovery control to remain in FACES rather than abort the job through a possible system error condition.

ANSI Standard Name File (ANSI)

The ANSI Standard Name File contains routine names used by AIR to support restrictions on Programmer defined names. The file is read by routine ATFILE. A fixed length sequential file of formatted records is used to provide names to be restricted. The first record indicates the file length and characteristics of the file records. The first record is read with a format of,

FORMAT (I4, 2X, A1, 2X, I4)

where first entry is the number of records

second entry is the type of each record entry

third entry is the length of elements expressed as the number of associated records.

The header record is followed by data records containing alphanumeric entries in A4 format, one entry per record. That is, the reading format is

FORMAT (A4)

The number of these records is established by the first header record count. The number of records associated as a data unit is indicated by the last count of the header record. For example, if the last count is 2, two sequential records are interpreted as a single datum.

For maintenance convenience, the data entries are stored in alphabetic order. This order is not required for proper operation.

Flag File (FLAG)

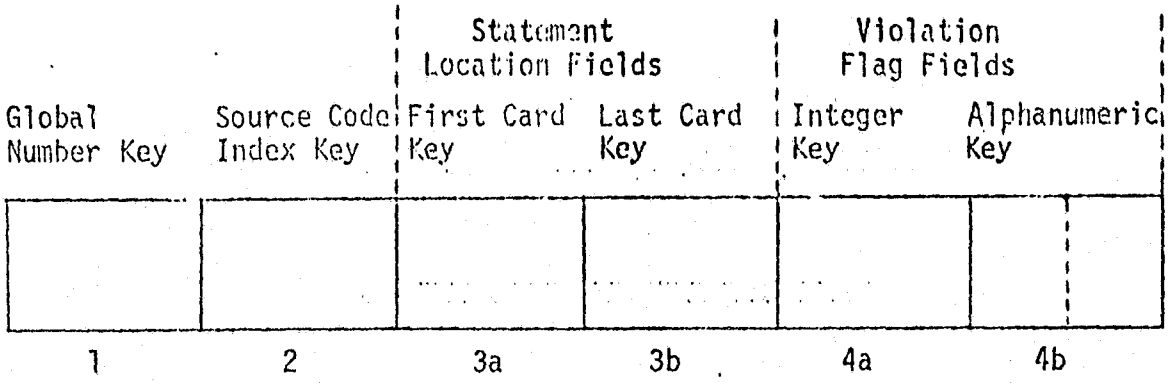
The Flag File is a variable length formatted sequential file containing diagnostic data for report generation. Records contain data created during FFE source code analysis and AIR investigations. Records contain both sort key information and report data to be processed by report generation.

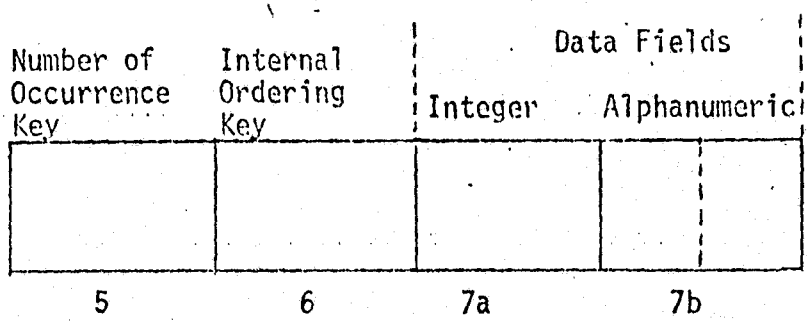
The contents of the Flag File are sorted prior to report generation. Sorting the contents causes all report items to become adjacent on the file. After the data has been consumed by the report process, the Flag File is set empty by a rewind/EOF operation.

Flag File may contain redundant information. Redundant information is not significant and is suppressed in report generation. Sorting records permits detection of redundant information.

Flag File creation requires cumulative addition to the file. As processing proceeds, new entries are added to the file contents to accumulate flags. For this reason, the Flag File is maintained with an end of file mark to delimit the data. When new entries are added, the EOF is destroyed, adding entries. The new file length is then sealed with an EOF mark.

Flag File (Sort File) Format





There are seven sort fields and three data fields. All are integer fields, except the alphanumeric Flag Field and the alphanumeric Data Field. Any field, alphanumeric or integer, which does not contain any information contains a zero. The output format is

FORMAT(5(2X,I5), 2X, 2A4, 3(2X,I5), 2X, 2A4)

1. Global Number Key - This sort key specifies whether the violation is to appear in the primary listing or in the secondary or display listings. A one in this key indicates that the violation is to

appear in the primary listing. An integer >1 indicates that the violation is to appear in the secondary or display listing.

2. Source Code Index Key - This sort key specifies the location of the beginning of the source code for the module in which the violation occurs. If the violation does not occur in a specific module, as in cyclic calls, then this key is zero.

3. Statement Location Fields - This area consists of two sort fields, the First Card Key and the Last Card Key.
   a. First Card Key - This sort key specifies the first card of the statement in which the violation occurs. If the violation does not occur in a specific card, as in cyclic calls, then this key contains a zero.
   b. Last Card Key - This sort key specifies the last card of the statement in which the violation occurs. If the violation does not occur in a specific card, as in cyclic calls, then this key contains a zero.

4. Violation Flag Fields - This area consists of two fields, an integer sort field and an alphanumeric non-sort field.
   a. Integer Violation Flag Key - This sort key specifies which violation has occurred. Each type of violation has its own integer code. If a violation is to appear within the primary listing or the secondary listing, this key specifies which.
   b. Alphanumeric Violation Code - This field contains the alphanumeric name of the violation.

5. Number of Occurrence Key - This sort key keeps track of the order in which violations of the same class occurred.

   If the query searches for local violations, then this sort key is set to zero each time a different module is examined, and incremented each time a violation occurs.

   If the query searches for global violations not involving path tracing (Parameter List Alignment and COMMON Block Alignment), then this sort key is set to zero when the query is invoked, and incremented each time a violation occurs.

   If the query searches for global violations using path tracing (Cyclic Call Search), this sort key is set to zero each time the query starts at a new path beginning.

6. Internal Ordering Key - This sort key specifies the internal ordering of the data concerning a violation. Since a violation may involve a great deal of information to be passed on to the user, this information must be placed in some order. For example, in a dimensional mismatch, the violation information would include (in the Data Fields) the name of the variable in violation, how many dimensions it has, and what those dimensions are. These pieces of data must retain their proper order for the output to be intelligible.

7. Data Fields - This area consists of two fields, an integer data field and an alphanumeric data field.

   a. Integer Data Field - This field contains integer data that describes or pinpoints the violation, such as the size of a COMMON Block.

b. Alphanumeric Data Field - This field contains alphanumeric data that describes or pinpoints the violation, such as the name of a variable.

Control Card File (CRTL)

The Control Card File is a variable length sequential file of 80 column card images. Each card image is a single FACES command card.

Since three phases are implemented for FACES, the control file should contain three files. Each file may be empty (i.e. only and EOF card) or contain commands appropriate to the pahse being executed. The command set for each phase is terminated by an EOF card.

FORTRAN Message File (FMSG)

For this implementation, the FORTRAN message file is equivalenced to the FLAG file. See Flag File description for characteristics.

Print File (PRNT)

The Print file is a standard formatted sequential file of 132 character positions. The file is assigned a variable name to assist in accommodating default file numbers on different systems for the principal output device. Formal file controls are not applied to print file operations.

Reserved Word File (RESW)

The Reserved Word File is a sequential file of formatted records containing FORTRAN "reserved words" for which restricted use is required. Data of this file is used by AIR to detect the use of restricted character strings used as Programmer defined elements. The file is read by routine ATFILE.

The file structure is identical to the ANSI file, composed of a header record describing the file followed by data entries. The header record format is,

FORMAT (I4, 2X, A1, 2X, I4)

Data records are recorded with format,

FORMAT (A4)

For maintenance convenience, alphabetic order of the data entries is used to store the file. This order is not required for proper operation.

Source Code Catalogue File (SCAT)

The Source Code Catalogue File is a formatted random access file of source code card images. The file is generated sequentially by the FFE from input source code card images. SCAT records are 80 column card images identical to the input source presented.

Through FFE analysis, the first and last cards of FORTRAN modules are identified. From this information, the origin of each module is developed and stored in the Directory for the module. The origin is a zero based absolute card image on SCAT. To retrieve a given relative card from the file, the absolute record number is computed as,

Absolute Record = Origin + Relative Card Number

Thus, the origin is actually the absolute card number of the card preceding the first module card (zero value for the first module).

Source code order on SCAT is identical to the order in which cards are presented to FACES. Thus, if source code is added to the system, the source is appended to the end of the current images.

Since SCAT is identical to the input source, if the file is lost, it can be reestablished by simply duplicating the source deck. Care should be taken to insure identical card images are created if this procedure is performed.

Special Note: Use of the value zero for the first module origin conflicts with the convention that zero implies an empty value. Since the first card image is nonzero, this value should be used rather than the origin to distinguish real modules from references.

Source Code Input File (SCIN)

The Source Code Input File is a formatted file of input source code card images. Each record of the file is an 80 column card image of FORTRAN source. A module set is delimited by an EOF mark. Several files can be presented at one time. One file of source code is processed for each ADD command.

Analysis Table File (TABL)

The Analysis Table File is an unformatted random access file for bulk storage of table data created by the FFE and AIR. A fixed number of records are allocated at the start of the file to store global data describing the software system under analysis. Global data records are followed by a series of Local Table records associated with individual modules presented for analysis.

The global data is composed of a global header and a series of global tables. The global header records the system status of the run creating the tables. Global data are tables which record the modules currently presented to the system and interaction among modules in the system. The Directory is created by the FFE as source code is presented for analysis. Other global tables are created by AIR to support the analysis of modules for indicated queries.

Module data is composed of a module header followed by the Local Tables produced by the FFE for the module. The local header contains entries indicating the active areas of table data (i.e., entries containing nonempty data). The Local Table records contain the full table space.

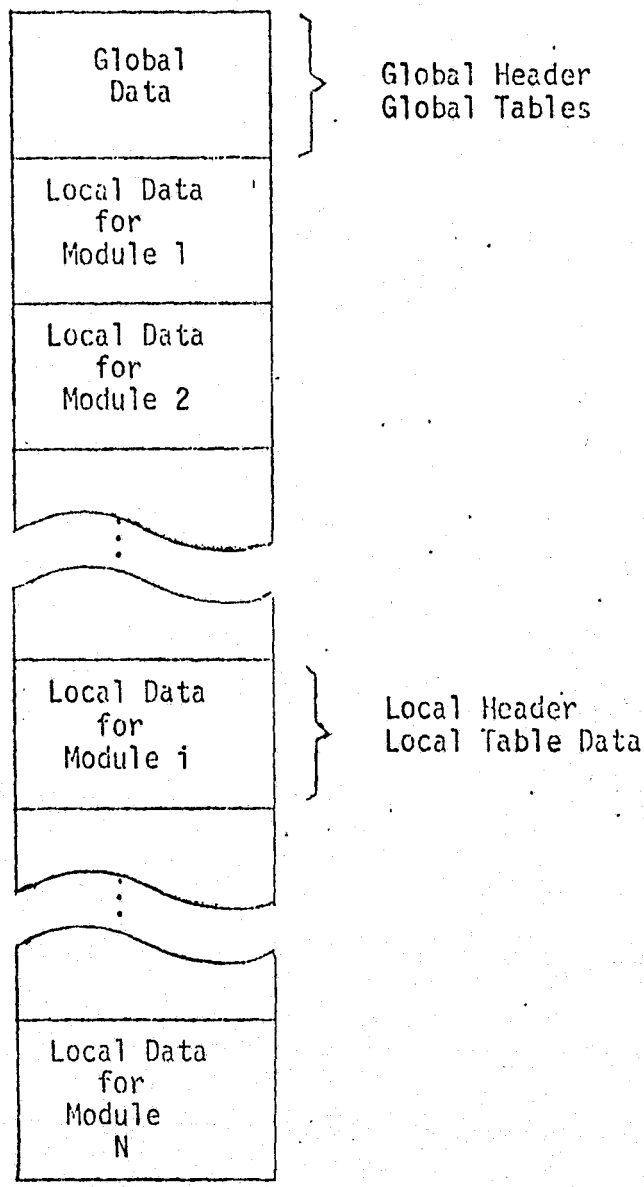
Module tables are accessed by the module number recorded in the Directory entry. Given the space reserved for Global data, the module number, and the space occupied by each module, the absolute record number of the Local Header can be computed. Table data follows the header record.

Global data is allocated by fixed records. These areas are specified by indicating the first record allocated for the global data. Global data

is stored in sequential records following the first record.

Record allocation is included in the COMMON block file description, /TABL/. The Table File can be reallocated by modifying the values set in the variables, however, current data files will become incompatable in the process.

Table File Structure



Global Table Allocation

Associated Common Block		Starting Record	Length (words)
/GHD/	GLOBAL HEADER	1	28
/DIR/	MODULE DIRECTORY	2	800
/SH/	SYSTEM HIERARCHY TABLE	10	400
/SHD/	SYSTEM HIERARCHY TO DIRECTORY TABLE	14	400
/IS/	INVERSE SYSTEM HIERARCHY TABLE	18	400
/ISD/	INVERSE SYSTEM HIERARCHY TO DIRECTORY TABLE	22	400
/COM/	COMMON BLOCK NAME TABLE	26	300
/LIN/	LINK LIST FOR COMMON NAMES TABLE	29	1000

Local Table File Structure for a Module

Associated Common Block		Starting Record Number	Length in Words
/MHD/	LOCAL HEADER	N	6
/SYM/ SYMTAB	MAIN SYMBOL TABLE	N+1	2800
/SYM/ SYMOVR	SYMBOL OVERFLOW TABLE	N+28	200
/USE/	USE TABLE	N+30	4000
/NOD/	NODE TABLE	N+70	2800
/SUC/	SUCCESSOR TABLE	N+98	1000
/PRE/	PREDECESSOR TABLE	N+108	1000

where N determined from module number entry of the Directory.

IX DATA STRUCTURES

Table of Contents

ALIGN (2,300) - (Alignment Tables)

The two Alignment Tables are temporary tables used during the COMMON Block Alignment Check and the Parameter List Alignment Check. All salient information concerning a COMMON Block or a Parameter List needed for an alignment check is packed into an Alignment Table. The model for the comparison is placed in Table 1, while the structure to be checked is placed in Table 2.

Each Alignment Table consists of one (1) column, which is one (1) computer word wide.

I. COMMON Block Alignment

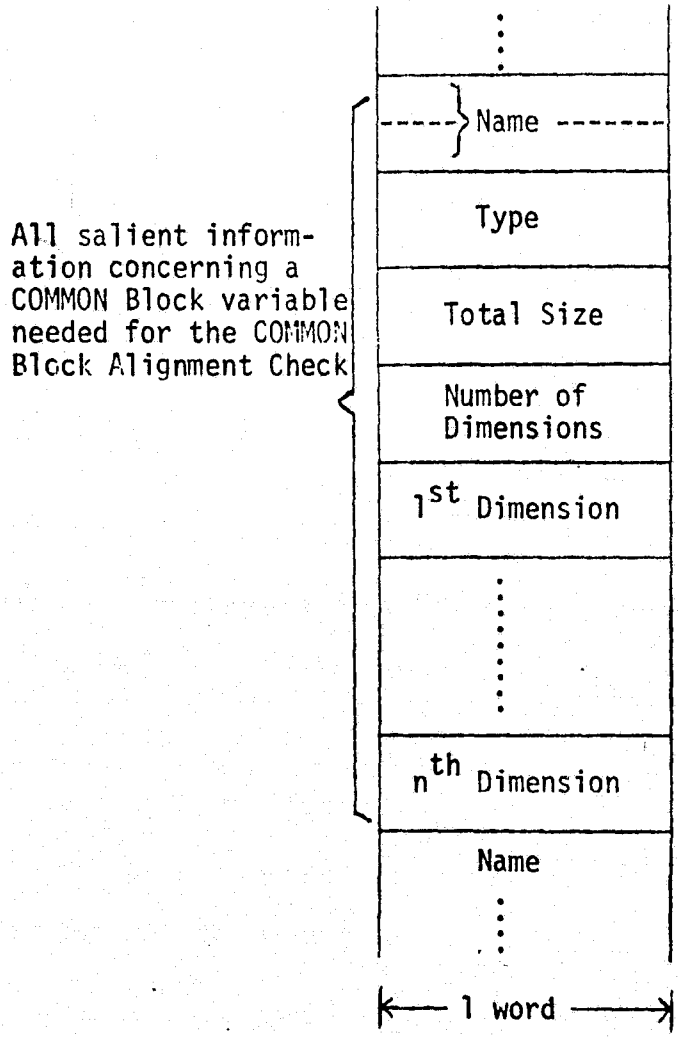


Table Entries for COMMON Block Variables

1) Name - (Alphanumeric Code)

Name is composed of two A4 format elements, which provide 8 characters for variable names.

2) Type - (Non-Negative Integer)

Type indicates the data type of the variable (see coding).

3) Total Size - (Positive Integer)

Total Size indicates the total number of computer words the variable has assigned to it. Being that different machines use a different number of words for different data types, Total Size is machine-dependent.

4) Number of Dimensions - (Non-Negative Integer)

Number of Dimensions indicates the number of dimensions the variable has, be it a scalar or an array.

5) k^{th} Dimensions - (Positive Integer)

The k^{th} Dimension is the integer value of the k^{th} dimension of the array variable. Scalar variables do not have this Alignment Table entry.

Note that the dimensions of an array appear in the Alignment Table in sequential order, with the left-most subscript of the array declaration being the first member of the sequential list.

Coding

Name

Value	Meaning
ALPHA ALPHA	Variable name in 2A4 format.

Type (as in the Type Codes of the Symbol Table)

Value	Meaning
0	Unknown
1	Floating Point
2	Double Precision
3	Complex
4	Logical
5	Neutral
6	Character Code (Hollerith)
7	Integer

Total Size

Value	Meaning
N	The compiler allocates the variable a total of N computer words

Number of Dimensions

Value	Meaning
0	The variable is a scalar.
N	The variable is an array and has N dimensions.

Coding (cont.)

k^{th} Dimension

Value	Meaning
N	The k^{th} dimension of the array is N.

II. Parameter List Alignment

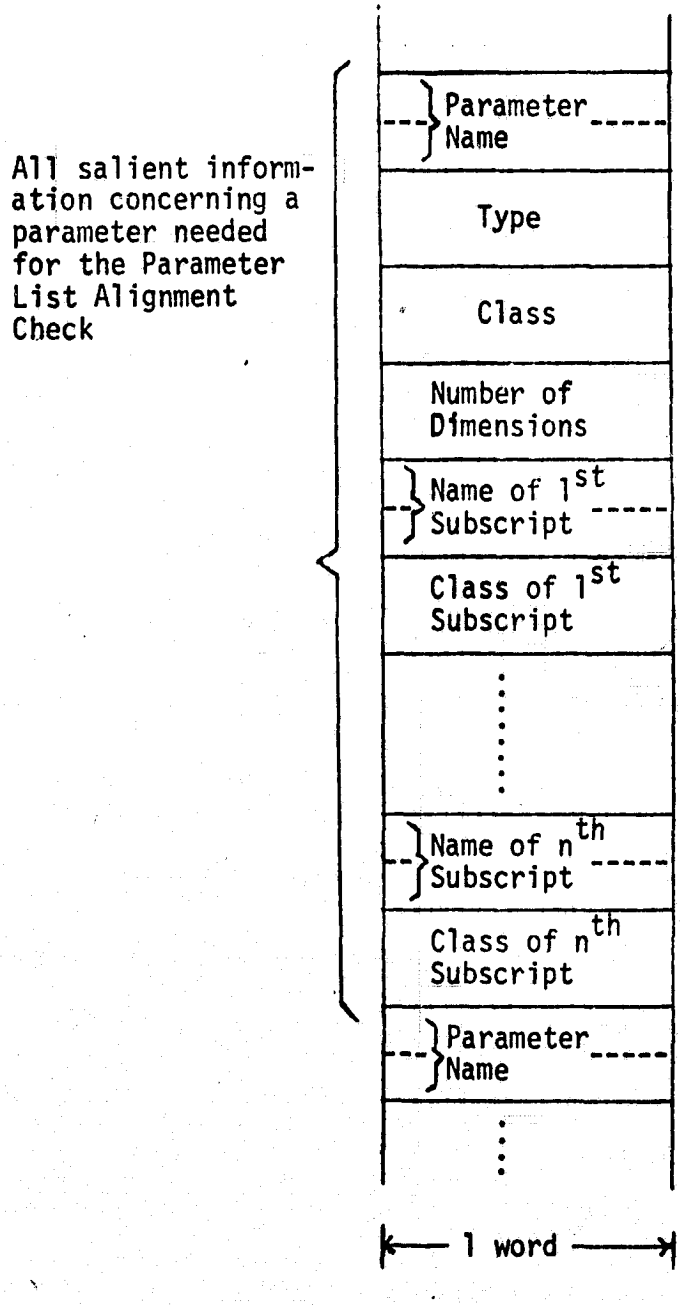


Table Entries for Parameter List Parameters

1) Parameter Name - (Alphanumeric Code)

Parameter Name is composed of two A4 format elements, which provide 8 characters for parameter names.

2) Type - (Non-Negative Integer)

Type indicates the data type of the parameter (see coding).

3) Class - (Non-Negative Integer)

Class indicates the class definition of the parameter (see coding).

4) Number of Dimensions - (Integer)

The absolute value of Number of Dimensions indicates the number of dimensions the parameter has, be it a scalar or an array.

5) Name of k^{th} subscript - (Alphanumeric Code)

This entry is composed of two A4 format elements, which provide 8 characters for the name of the parameter's k^{th} subscript. Only parameters which are arrays have this Alignment Table entry.

6) Class of k^{th} subscript - (Non-Negative Integer)

This entry indicates the class definition of the parameter's k^{th} subscript (see coding). Only parameters which are arrays have this Alignment Table entry.

Note that descriptions of an array's subscripts appear in Alignment Table in sequential order, with the array's left-most subscript being the first member of the sequential list.

Coding

Parameter Name

Value		Meaning
ALPHA	ALPHA	Parameter Name. The parameter is not an arithmetic subexpression.
*SUB	EXPR	The parameter is an arithmetic subexpression.

Type (as in the Type Codes of the Symbol Table)

Value	Meaning
0	Unknown
1	Floating Point
2	Double Precision
3	Complex
4	Logical
5	Neutral
6	Character Code (Hollerith)
7	Integer

Coding (cont.)

Class (as in the Class Codes of the Symbol Table)

Value	Meaning
0	Unknown
1	Subroutine Name
2	Statement Function Name
3	Array Variable
4	Function Name
5	Statement Label
6	Scalar Variable
7	Common Block Label
8	Constant
9	Entry Point Name
10	
11	
12	Program Name
13	Temporary Variable
14	
15	Statement Function Dummy Parameter
16	Explicit External Function or Subroutine

Number of Dimensions

Value	Meaning
-N	The parameter is an array having N dimensions, but appears in the parameter list without any subscript. This is the only place a negative value occurs in AIR.
0	The parameter is not an array.
N	The parameter is an array having N dimensions.

Coding (cont.)

Name of k^{th} Subscript

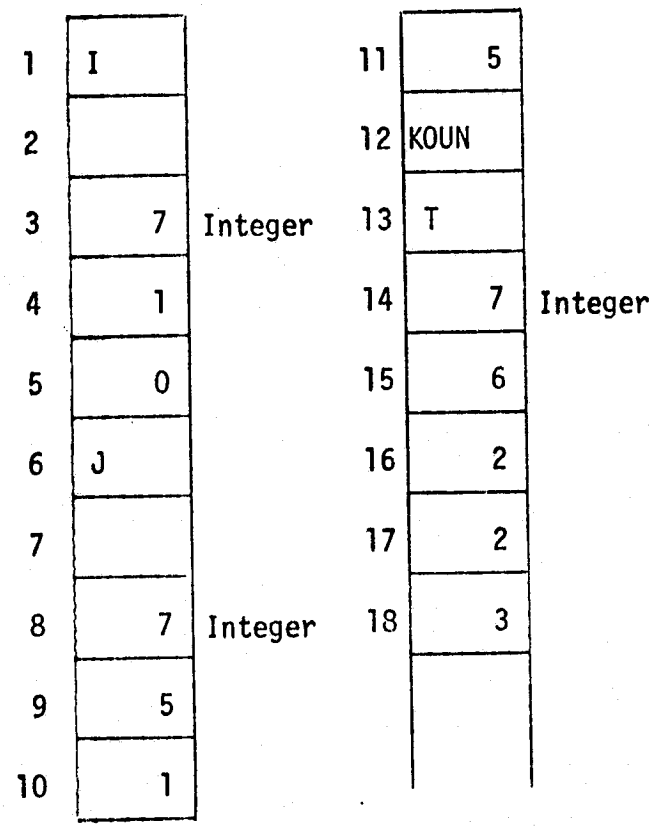
Value	Meaning
ALPHA ALPHA	Subscript name. The subscript is not an arithmetic subexpression.
*SUB EXPR	The subscript is an arithmetic subexpression.

Class of k^{th} Subscript (as in the Class Codes of the Symbol Table)

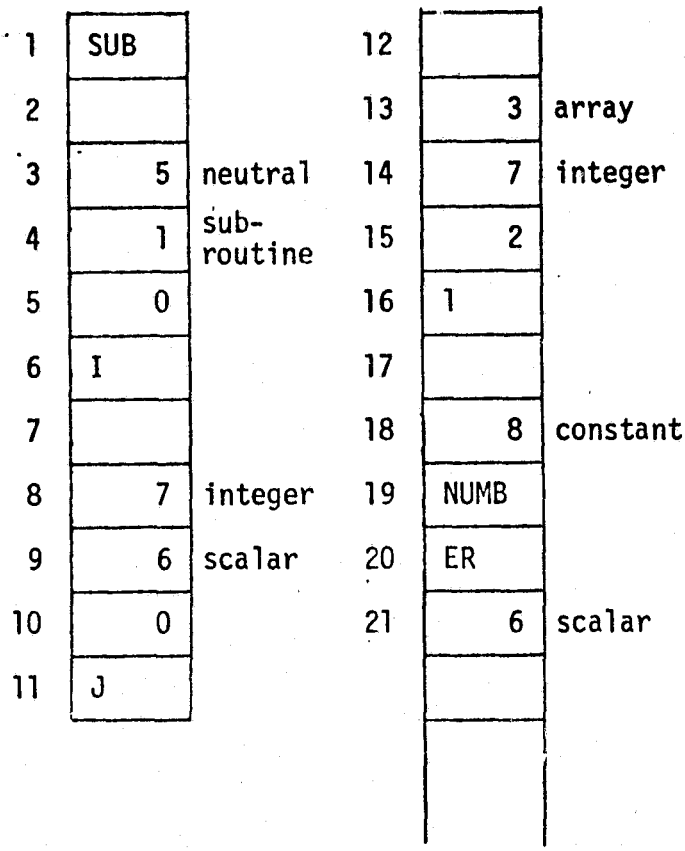
Value	Meaning
0	Unknown
1	Subroutine Name
2	Statement Function Name
3	Array Variable
4	Function Name
5	Statement Label
6	Scalar Variable
7	Common Block Label
8	Constant
9	Entry Point Name
10	
11	
12	Program Name
13	Temporary Variable
14	
15	Statement Function Dummy Parameter
16	Explicit External Function or Subroutine

Assume all variables are of type integer and occupy one word of memory:

Example 1) COMMON/A/I,J(5), KOUNT(2,3)

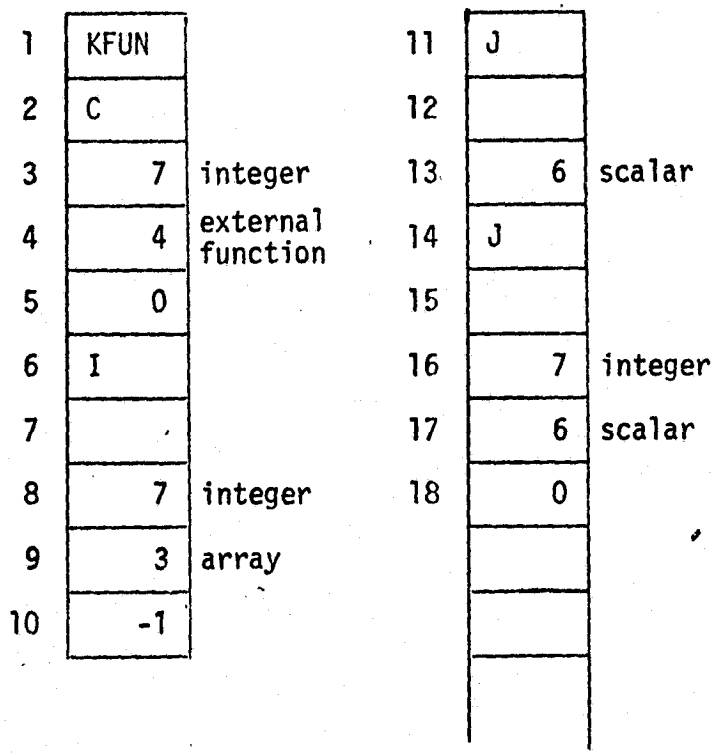


Example 2) CALL SUB(I,J(1,NUMBER))



Example 3) FUNCTION KFUNC(I,J)

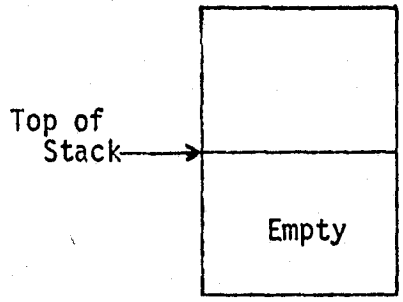
DIMENSION I(J)



BEGIN/END USE CODE STACKS

LSTSTK(10) (List Use Code Stack)

SBESTK(10) (Subexpression Use Code Stack)



Begin/End Use Stack

The List Use Code Stack and Subexpression Use Code Stack have the same structure. They are distinguished by function only. Begin/End stacks are used to record USE Table positions in which Begin USE codes were entered permitting connection of associated End USE codes within the same statement.

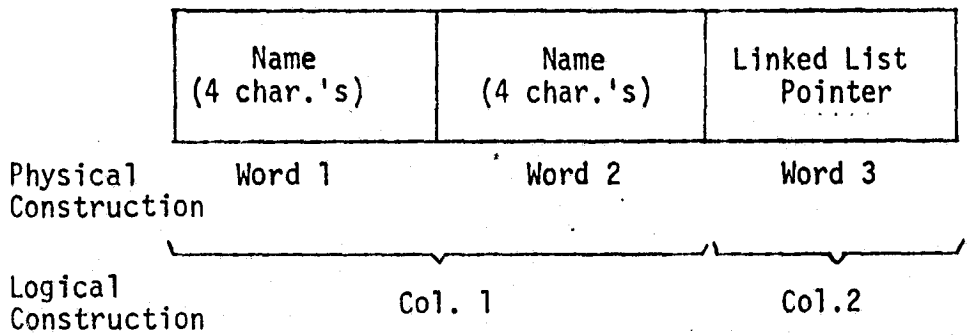
When the Begin USE code is recorded, the USE Table position is placed on the top of the stack. If another Begin code appears, the stack is pushed down one level and the new USE Table position recorded.

When an End USE code appears, it is associated with the top Begin stack entry. Linkage among the USE Table entries is established and the top element removed from the stack.

Stack size determines the level of nesting permitted by intra-statement list structures. Current use of structures is small compared to allocated nesting.

COMMON BLOCK REFERENCE TABLES

COMTAB(3,100) - (COMMON Block Name Table)



The COMMON Block Name Table is a sequential list containing COMMON Block names and pointers to the Linked List of COMMON Blocks Table. The COMMON Block Name Table records the name of the COMMON Blocks which appear in the source code.

Each COMMON Block Name Table Entry consists of three (3) computer words organized as two (2) logical columns.

Entry Contents - (Logical Columns)

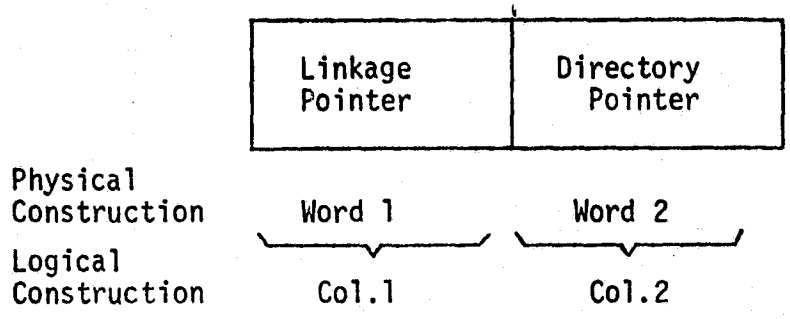
1. COMMON Block Name - (Alphanumeric Code)

   The COMMON Block name is composed of two A4 format elements, providing 8 characters for COMMON Block names.

2. Linked List Pointer - (Positive Integer)

   The Linked List Pointer points to the first member of a linked list in the Linked List of COMMON Blocks Table.

LINTAB (2,500) - (Linked List of COMMON Blocks Table)



The Linked List of COMMON Blocks Table is a linked list containing pointers to the Directory. The Linked List of COMMON Blocks Table records the modules in which a COMMON Block occurs.

Each Linked List of COMMON Block Table Entry consists of two (2) computer words organized as two (2) logical columns.

Entry Contents - (Logical Columns)

1) Linkage Pointer - (Non-Negative Integer)

   The Linkage Pointer either points to the next entry in the linked list or indicates that this entry is the last entry in the linked list.

2) Directory Pointer - (Positive Integer)

   The Directory Pointer points to a module in the Directory in which the COMMON Block (specified in the COMMON Block Name Table) occurs.

Coding

Linkage Pointer

Value	Meaning
0	Last member of linked list.
P	Pointer to next member of linked list.

Directory Pointer

Value	Meaning
DP	The name of a module which contains the COMMON Block is in row DP of the Directory.

Example:

```
SUBROUTINE A
COMMON/X/.....
COMMON/Y/.....
        :
SUBROUTINE B
COMMON/X/.....
COMMON/Z/.....
        :
FUNCTION C(Q,R)
COMMON/X/.....
COMMON/Z/.....
        :
```

COMMON BLOCK NAME TABLE

COMMON BLOCK Name	Pointer to Linked List
X	1
Y	2
Z	4

LINKED LIST TABLE

Index	Linkage Pointer	Pointer to Directory
1	3	4
2	0	1
3	5	4
4	6	6
5	0	4
6	0	6

DIRECTORY

Index	Module Names		
1	B		
2			
3			
4	A		
5			
6	C		

MSTR(20), TSTR(20), LSTR(20), PSTR(20) - (The Control Stack)

Module Number Stack Register	Table Name Stack Register	List Indicator Stack Register	Pointer Stack Register
Stack 1	Stack 2	Stack 3	Stack 4

The Control Stack is a stack consisting of four single-column stacks. It keeps track of which elements of which lists in which tables of what modules are currently being examined by AIR. The top of the Control Stack contains information about the table currently being examined by the AIR subsystem.

Each single-column stack entry consists of one (1) computer word.

Entry Contents - (Single-Column Stacks)

1) Module Number Stack Register - (Non-Negative Integer)

The top of this stack contains the module number of the module that was last brought into main memory by AIR.

2) Table Name Stack Register - (Alphanumeric Code)

The top of this stack contains the table currently being examined by AIR. It consists of one A4 format element which provides 4 characters for table names.

3) List Indicator Stack Register - (Non-Negative Integer)

The top of this stack contains the list indicator for the list (in the table indicated by the Table Name Stack Register) currently being examined by AIR.

The positive value in the list indicator may either

a) refer to the length of the non-traversed part of the list if the list is sequential and its overall length is known, or

b) merely indicate that some other type of list, such as a linked list, has not been completely traversed.

4) Pointer Stack Register - (Positive Integer)

The top of this stack points to the row (in the table indicated by the Table Name Stack Register) currently being examined by AIR. It is equal to the current row pointer to the table indicated by the Table Name Stack Register.

Coding: for top of the Stack entries.

Module Number Stack Register

Value	Meaning
0	A module has not yet been brought into main memory.
N	The module having module number N currently resides in main memory.

Table Name Stack Register

Value	Meaning
ALPHA	Name of table currently being examined by AIR.

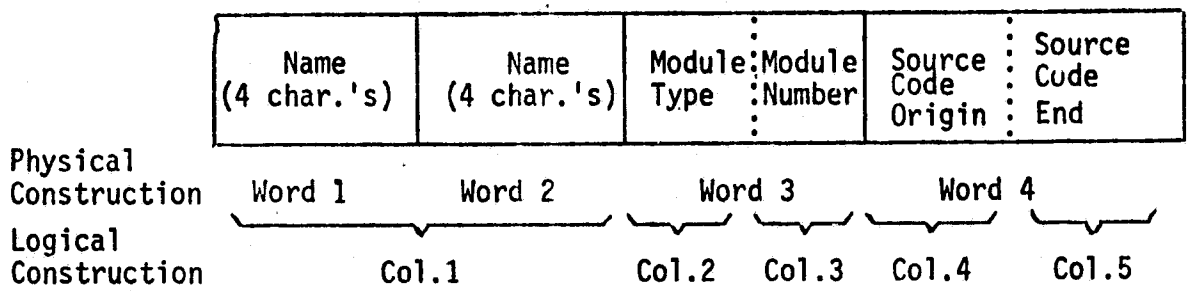
List Indicator Stack Register

Value	Meaning
0	List has been completely traversed
N	List has not been completely traversed.

Pointer Stack Register

Value	Meaning
N	Row N in the table indicated by the Table Name Stack Register.

DIREC (4,200) - (Directory)



The Directory is a sequential table containing module names in alphabetical order, module type, and access information to retrieve tables and source code associated with the module.

Each Directory entry is composed of four (4) physical computer words containing five (5) logical columns of information.

Entry Contents (Logical Columns)

1) Module Name - (Alphanumeric Code)

The module name is composed of two A4 format elements providing 8 characters for module names.

2) Module Type - (Positive Integer)

Module Type indicates the type of module the name defines (see Coding).

3) Module Number - (Non-Negative Integer)

Module numbers are assigned to modules as they are defined by incoming source text. The module number provides location key information for obtaining local tables from the random local table file.

4) Source Code Origin - (Non-Negative Integer)

The Source Code Origin entry indicates the key to the first source code card image on the source code catalogue.

5) Source Code End - (Non-Negative Integer)

The source code end contains a value indicating how many source card images are recorded for the module.

Coding

The following codes are assigned to Directory entries:

Module Name

Value	Meaning
0	Empty Directory entry
ALPHA	Module name.

Module Type

Value	Meaning
1	Program
2	Subroutine
3	Function
4	Block Data
5	Secondary Subroutine Entry Point
6	Secondary Function Entry Point.

Module Number

Value	Meaning
0	No tables for module exist. Source code for this module not yet received.
N	Index value used to compute local tables for module on random table file.

Source Code Origin

Value	Meaning
0	No source text available.
N	Absolute card image number of the card preceding the first module card on source catalog.

Source Code End

Value	Meaning
0	No source text available.
N	Module contains N card images.

Remarks:

If the module type is a secondary entry point, the Module Number, Source Code Origin, and Source Code End entries will be duplicates of the primary entry point values.

MATCH(51) - (FORTRAN Key Word Match Table)

The FORTRAN Key Word Match Table contains the leading four characters of FORTRAN statement key words set in BLOCK DATA. This read-only table is used in parsing FORTRAN statements and establishing Statement Type codes for Node Table entries.

Given the leading four characters of a key word, the table is interrogated in the following fashion:

1. The third and fourth characters of the presented word are extracted and converted to values between 0 and 26. The values 1 through 26 correspond to the alphabetic characters A through Z. The value 0 represents all other characters.
2. Initial entry into the table is accomplished by hash coded access using the values developed. The hash formula used is:

   entry = VAL3 + VAL4 + 10

   where VAL3 and VAL4 are the alphabetic indices developed from characters 3 and 4.
3. If the entry address is empty or outside the table range, no match is found.
4. If the entry address matches the presented characters string, a match is found.
5. If the entry position is occupied by a nonmatching entry, searching is diverted to the list position indicated by BIAS, where a sequential search is performed until a match is found or an empty position is discovered (no match).

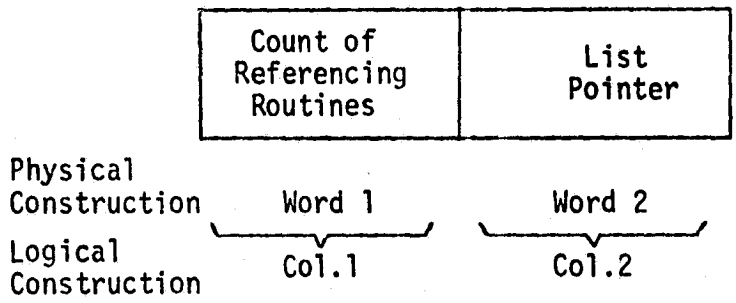
EXAMPLES:

1. Keyword DATA. Converting the T and A to numerical indices produces values 20 and 1 respectively. Inserting these into the formula produces the entry value 20 + 1 + 10 = 31. The character string matches the entry at position 31.
2. Keyword FORMAT. Converting R and M and applying the entry formula produces 18 + 13 + 10 = 31. Comparing the string FORM to the entry does not match (entry 31 contains DATA). A search is diverted to position 3 for a sequential search. Position 3 contains the character string FORM producing a match.
3. Keyword PARAMETER. Converting R and A applying the formula produces an entry address 18 + 1 + 10 = 29. Since table position 29 is empty (value 0), no match is found.
4. Keyword IF. Since characters 3 and 4 are both blank, the entry address for IF is 0 + 0 + 10, a matching location in the table.

Design Considerations. The odd structure of MATCH merits some explanation. The table is structured to minimize table searches. The mapping formula usually permits immediate access to matching keyword positions with the first entry. Since direct matching entries are tightly clustered, the offset of 10 produces empty table space in which to place collision entries.

The use of a BIAS (collision area origin) equal to 3 is historical; exact reasoning for this choice is unknown. Since table indices are used for statement type codes, the original table structure was maintained.

ISTAB (2,200) - (Inverse System Hierarchy Table)



The Inverse System Hierarchy Table is a sequential table which is basically a horizontal extension of the Directory. Along with the Inverse System Hierarchy to Directory Table (ISDTAB), this table records the called-by hierarchy of the software system being analyzed.

Each Inverse System Hierarchy Table Entry consists of two (2) computer words organized as two (2) logical columns.

Entry Contents - (Logical Columns)

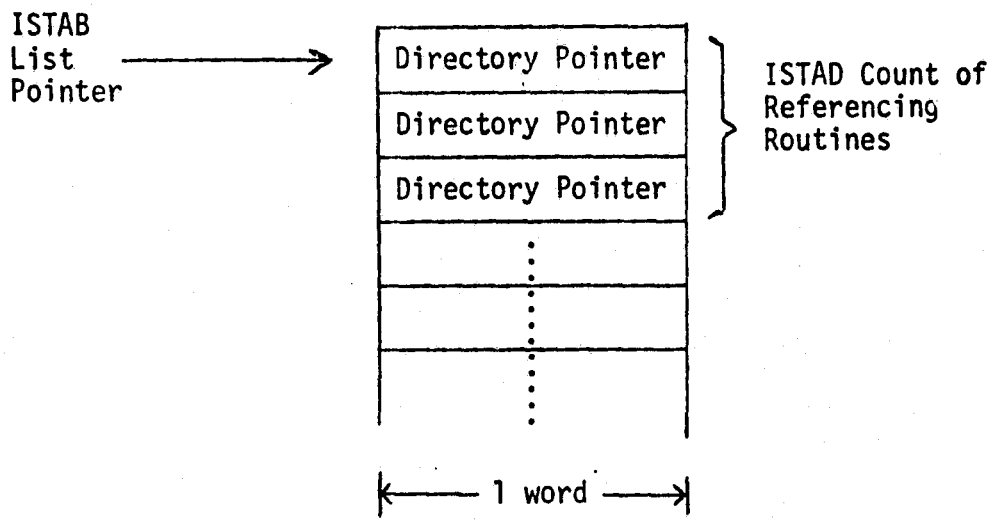
1) Count of Referencing Routines - (Non-Negative Integer)

   This is a count of the number of different routines the routine specified in the Directory is referenced by. It is the length of the associated list in the Inverse System Hierarchy to Directory Table.

2) List Pointer - (Non-Negative Integer)

   The List Pointer points to the first member of a list in the Inverse System Hierarchy to Directory Table.

ISDTAB (400) - (Inverse System Hierarchy to Directory Table)



The Inverse System Hierarchy to Directory Table is a set of sequential lists whose structure is defined by ISTAB. Along with the Inverse System Hierarchy Table, this table records the called-by hierarchy of the software system being analyzed.

Each Inverse System Hierarchy to Directory Table Entry is one (1) computer word wide. Entry into a list of referencing routines is by way of the ISTAB List Pointer. The length of a list is determined by the ISTAB Count of Referencing Routines.

Entry Contents - (Positive Integer)

Each list entry points to a module in the Directory.

Example:

```
SUBROUTINE C
     :
     :
CALL SUB
     :
     :
SUBROUTINE D
     :
     :
CALL SUB
     :
     :
SUBROUTINE SUB
```

Directory

Index			
1			
2	SUB	5	205
3			
4			
5	C	6	206
6	D	3	203

Inverse System Hierarchy

2	3

Inverse System Hierarchy to Directory

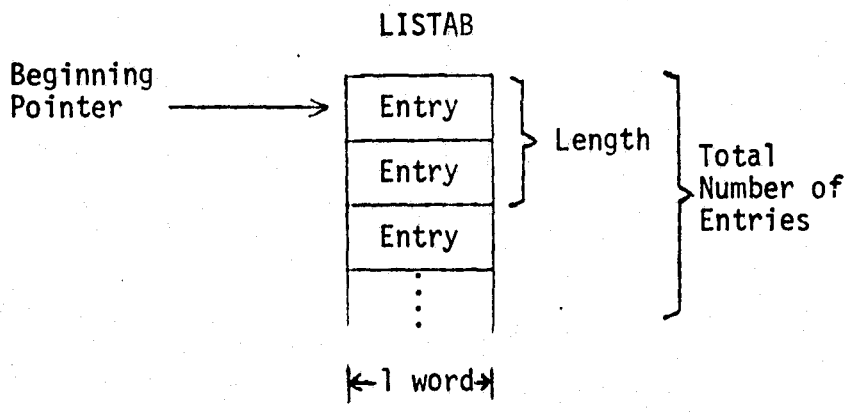
Index	
1	
2	
3	5
4	6

LISTAB (500) - (List Table)

MAP(6,20) - (List Table Map)

MAP ENTRY

Total Number of Entries	Number of Remaining Entries	Type	Length	Beginning Pointer	Current Pointer
Word 1	Word 2	Word 3	Word 4	Word 5	Word 6



The List Table Map describes sequential lists in the List Table. The n^{th} description in the MAP, i.e., the n^{th} row, refers to the n^{th} list in the List Table. Thus, the List Table's structure is defined by the List Table Map.

A List Table Map Entry consists of six (6) elements, each consisting of one (1) computer word. Each List Table Entry consists of at least one (1) computer word.

Entry Contents - (Words)

List Table Map

1) Total Number of Entries - (Non-Negative Integer)

This element specifies the total number of entries there are in the n^{th} list in the List Table.

2) Number of Remaining Entries - (Non-Negative Integer)

This element specifies the number of entries in the list that have not been examined yet. It can also be described as the length of the list that has not as yet been traversed.

3) Type - (Alphanumeric Code)

This element specifies the type of the entries stored in the n^{th} sequential list.

4) Length - (Positive Integer)

This element specifies the length, in computer words, of each entry in the n^{th} sequential list.

5) Beginning Pointer - (Positive Integer)

This element points to the beginning of the n^{th} sequential list.

6) Current Pointer - (Non-Negative Integer)

This element points to the entry in the list which is currently being examined.

List Table

Each entry of the List Table is a member of a sequential list whose structure is defined by the List Table Map. The contents of each list are independent of each other.

Coding

List Table Map

1) Total Number of Entries

Value	Meaning
0	The list is empty, i.e., a "null" list.
N	The list has N entries.

2) Number of Remaining Entries

Value	Meaning
0	The list has been completely examined.
N	The last N entries in the list are still to be examined.

3) Type

Value	Meaning
A	The list contains alphanumeric information.
I	The list contains integer information.

4) Length

Value	Meaning
N	Each entry in the list consists of N computer words.

5) Beginning Pointer

Value	Meaning
BP	The first entry in the list is at row BP in the List Table.

6) Current Pointer

Value	Meaning
0	The examination of the list has not yet begun.
CP	The list entry at row CP is currently being examined.

Example:

List 1: ABLE BAKER, CHARLIE

List 2: 18, 16, 14, 10

Case 1: The lists have been loaded into the List Table and their descriptions into the List Table Map, but no other processing has yet occurred.

List Table Map

Index	Total No. of Entries	No. of Remaining Entries	Type	Length	Beginning Pointer	Current Pointer
1	3	3	A	2	1	0
2	4	4	I	1	7	0

List Table

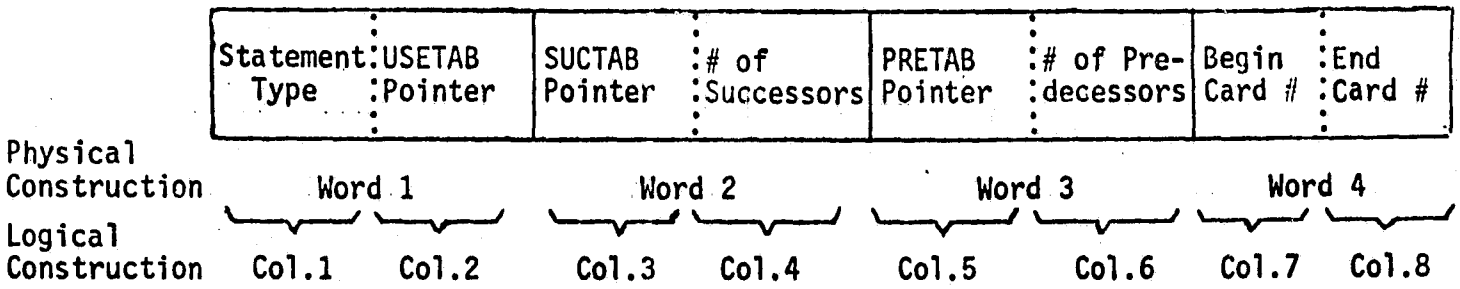
Index	
1	ABLE
2	
3	BAKE
4	R
5	CHAR
6	LIE
7	18
8	16
9	14
10	10
11	
12	

Case 2: The MANL (Manipulate List Table Map) subroutine has been called once for List 1 and thrice for List 2.

List Table Map

Index						
1	3	2	A	2	1	3
2	4	1	I	1	7	9

NODTAB (4,700) - (Node Table)



The Node Table is a sequential list characterizing each non-comment source statement. The position of the Node Table entry corresponds to the position of the source statement in Node Table. For example, source statement number 5 will be characterized by Node Table entry 5.

Node Table entries identify the type of source statement and access information for the symbolic element uses by that statement. Graphical flow of control generated by the statement is found through immediate predecessor/successor relations recorded in the Predecessor Table and Successor Table. The source code statement can be retrieved from the source code catalogue using the first/last relative card pointer.

Each Node Table entry is composed of four (4) computer words organized as eight (8) logical columns.

Entry Contents (Logical Columns)

1) Statement Type - (Non-Negative Integer)

The statement type is an integer code indicating the type of source statement encountered.

2) USETAB Pointer - (Non-Negative Integer)

The USETAB pointer is an integer value indicating the first Use Table entry for this statement.

3) SUCTAB Pointer - (Non-Negative Integer)

The SUCTAB Pointer is an integer index to the first immediate successor of this statement.

4) Number of Successors - (Non-Negative Integer)

The number of successors indicates a count of SUCTAB entries which belong to this statement.

5) PRETAB Pointer - (Non-Negative Integer)

The PRETAB Pointer is an integer index to the first immediate predecessor of this statement.

6) Number of Predecessor - (Non-Negative Integer)

A count of immediate predecessors to this statement.

7) Begin Card Number - (Positive Integer)

Card image count of the first source card on which this statement begins. The count is relative to the first card of the source code module.

8) End Card Number - (Positive Integer)

Card image count of the last source card on which this statement ends. The count is relative to the source module origin.

Coding

The following codes are assigned to the Node Table entries:

Statement Type

Value	Meaning
0	Empty entry - no statement exists
3	FORMAT
4	PRINT
5	IMPLICIT type declaration
6	NAMELIST
7	ENCODE
8	DECODE
9	PUNCH
10	IF
11	COMPLEX type declaration
12	EXTERNAL
13	BLOCK DATA
14	END
15	READ
20	ENDFILE
23	REAL type declaration
24	BACKSPACE
26	LOGICAL type declaration
27	FUNCTION
28	DIMENSION

Statement Type (cont'd)

Value	Meaning
30	SUBROUTINE
31	DATA
32	PROGRAM
33	DOUBLE PRECISION type declaration
34	CALL
35	INTEGER type declaration
36	COMMON
38	ASSIGN
39	WRITE
40	EQUIVALENCE
41	STOP
42	REWIND
44	CONTINUE
45	GOTO
48	ENTRY
50	PAUSE
51	RETURN
52	Assignment statement
53	DO statement
54	Statement function
99	Unrecognized statement

(Note: Integer values between 1 and 99 which do not appear are unused codes. These values will not occur in Statement Type Entries.)

USETAB Pointer

Value	Meaning
0	No uses recorded for this statement
P	List of uses for this statement begins in position P of USETAB.

SUCTAB Pointer

Value	Meaning
0	No successors
SP	The list of statement's immediate successors begins in position SP of SUCTAB.

Number of Successors

Value	Meaning
0	No successors
M	This statement has M successors. Immediate successors are found in SUCTAB positions SP through (SP+M-1).

PRETAB Pointer

Value	Meaning
0	No predecessors
PP	The list of statement's immediate predecessors begins in position PP of PRETAB.

Number of Predecessors

Value	Meaning
0	No predecessors
N	This statement has N predecessors. Immediate predecesssors are found in PRETAB positions PP through (PP+N-1).

Beginning Card Numbers

Value	Meaning
MC	This source statement begins on card image MC relative to source module origin. The first card of the module is assigned a card image count of 1.

Ending Card Number

Value	Meaning
NC	This source statement ends on card image NC relative to source module origin.

Remarks

Card Numbers

In general, $MC_i \leq NC_i$ for a particular source statement.

Condition	Meaning
$MC_i = NC_i$	Source statement occupies one card image.
$MC_i < NC_i$	Source statement occupies $(NC_i - MC_i + 1)$ card image.

For two adjacent statements, $NC_i + 1 \leq MC_{i+1}$.

Condition	Meaning
$1 + NC_i = MC_{i+1}$	No comments between statements i and i+1
$1 + NC_i < MC_{i+1}$	Comment cards between statements i and i+1.

If multiple statements appear on a single card image, values of NC and MC will be identical for the statements.

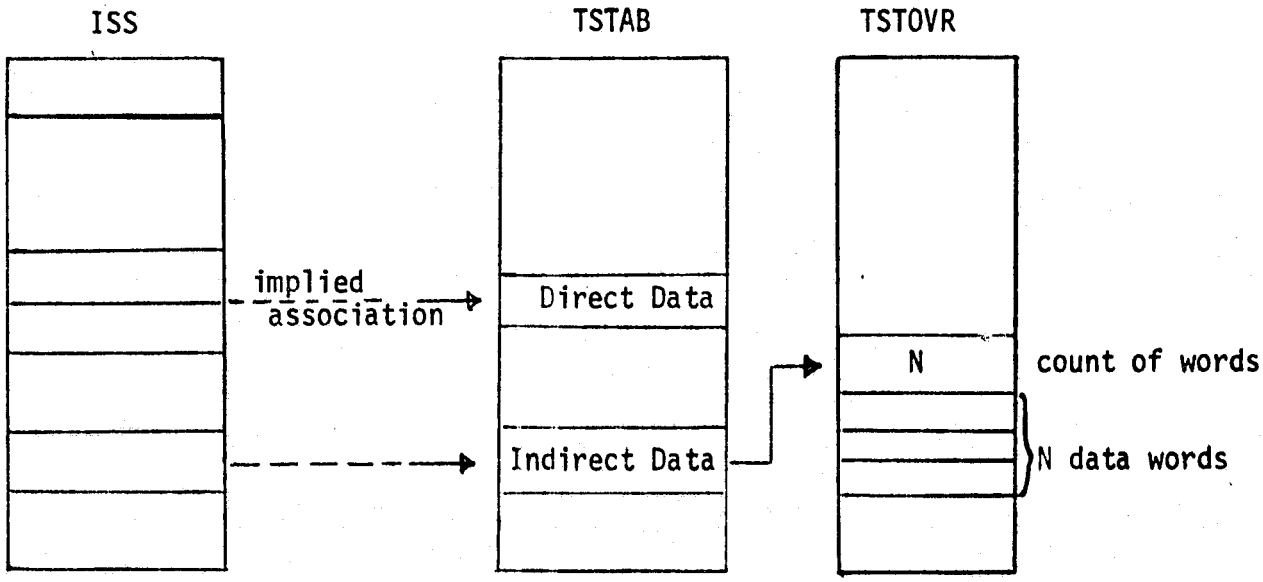
If comments appear between continuation cards of a single statement, the comment cards are catalogued as part of the source statement. Comment images for this case are included in the source statement and counts.

Parsing Tables

ISS (400) (Intermediate Symbol String)

TSTAB (2,300) (Temporary Symbol Table)

TSTOVR (100) (Temporary Symbol Overflow Table)



The Parsing Tables are composed of an Intermediate Symbol String structure and associated Temporary Symbol Table data. The Parsing Tables are constructed by the Scan process to normalize the presentation of source text to the Parsing routines. One FORTRAN statement (with continuation cards) is presented at a time.

The Intermediate Symbol String (ISS) is a sequential list characterizing statement entries found on the FORTRAN source code card image. Entries of ISS classify the lexical items to be parsed. Single character strings are also stored in ISS entries. The last entry in the list is an End of Statement code.

Where multiple character strings are present, an ISS code entry has implied association with a Temporary Symbol entry containing the characters. If the character string is short enough to fit in the main Temporary Symbol Table, the string is packed into the entry. If the character string is too long for the Temporary Symbol main entry, the character string is placed in the overflow data space and a pointer to the character string structure is made.

Entry Contents

ISS code - (A Format character code)

Represents a character string classification of the associated TSTAB entry, special characters found in the FORTRAN text, or an End of statement code.

TSTAB Entry - (A Format characters or non-negative integer pointers)

Holds either the direct character string of FORTRAN lexical items or an indirect pointer to the character string in the Overflow table.

TSTOVR - (Data structure of a count followed by character data)

Holds oversized lexical character strings. The length of the data structure is indicated by the first entry.

Coding

ISS entries

Value	Meaning
0	empty entry
V	Alphanumeric character string
I	Numeric character string
F	Floating Point constant
D	Double precision floating point constant
C	Complex constant
H	Character literal string (Hollerith)
T	Logical constant
R	Relational Operator
L	Logical Operator
=	FORTRAN separators
(	
)	
,	
.	
$	
'	
"	
+	FORTRAN arithmetic operator symbols
-	
*	
/	
$$	End of statement code

TSTAB Entries

ISS Code	Value	Meaning
any	0 0	Empty
V	ALPHA ALPHA	8 characters or less of alpha-numeric character string in 2A4 format
I, F, D, C	ALPHA ALPHA	Direct character string data of 8 characters or less in 2A4 format
H	0 P	Indirect pointer to character string data structure held in overflow table.
all others		No TSTAB entry associated

TSTOVR Entries

Value	Meaning
0	Empty
N	Positive integer indicating length of character string in words
ALPHA	Character string stored in A4 format

Special Notes: Notice that not all ISS entries have associated TSTAB entries. Only ISS entries corresponding to Operands (i.e., variables and constants) and FORTRAN keyword text are recorded in TSTAB. Since some ISS entries do not have character strings stored, by convention the pointer to TSTAB leads the pointer to ISS by one position. For example, when a V entry of ISS is encountered, TSTAB is currently pointing to the appropriate character string entry. When both are advanced, TSTAB is positioned to the next character string entry associated with an ISS code.

The leading convention permits valid TSTAB pointer values to extend one entry beyond the last nonempty entry pointer. This extra position is permitted where special characters terminate ISS entries. Since there are no further references to character string codes ISS entries, the actual pointer value and table entry are not significant in processing.

A = 3 + (C(I,J) / 2**N)

INTERMEDIATE SYMBOL STRING
LENGTH = 400
LAST ENTRY = 18
CURRENT POINTER = 1
ZERO EQUAL SIGN = 8

TEMPORARY SYMBOL TABLE
LENGTH = 10
LAST ENTRY = 7
CURRENT POINTER = 1

OVERFLOW LENGTH = 100
LAST ENTRY = 0
CURRENT POINTER = 0

	INDEX	CONTENTS
**	1	V
	2	=
	3	V
	4	+
	5	(
	6	V
	7	(
	8	V
	9	,
	10	V
	11	)
	12	/
	13	I
	14	*
	15	*
	16	V
	17	)
	18	$$

	INDEX	CONTENTS	ADDRESS	OVERFLOW CONTENTS
**	1	A		
	2	B		
	3	C		
	4	I		
	5	J		
	6	2		
	7	N		

```
      DIMENSION Q(2,4), P(5,5,5)

NTERMEDIATE SYMBOL STRING
ENGTH =   400
AST ENTRY =      17
URRENT POINTER =       1
RD EQUAL SIGN =       0

NDEX   CONTENTS
    1   V
    2   V
    3   (
    4   I
    5   ,
    6   I
    7   )
    8   ,
    9   V
   10   (
   11   I
   12   ,
   13   I
   14   ,
   15   I
   16   )
   17   $$

TEMPORARY SYMBOL TABLE
LENGTH =   500
LAST ENTRY =       3
CURRENT POINTER =       1

OVERFLOW LENGTH =   100
LAST ENTRY =       0
CURRENT POINTER =       0

 INDEX   CONTENTS    WORDS   OVERFLOW CONTENTS
**    1   DIMENSION
      2   NC
      3   2
      4   4
      5   3
      6   5
      7   5
      8   5
```

ORIGINAL PAGE IS
OF POOR QUALITY

```
CARD NUMB   STMT NUMB      LABEL *SOURCE TEXT
         1           1            COMPLEX FUNCTION CHECK1 (ZINPUT, MULT)
         2           1      C*************************************************************C
         3           1      C   THIS ROUTINE PROVIDES TESTS FOR THE FOLLOWING STATEMENT TYPES :     C
         4           1      C      EQUIVALENCE STATEMENTS                                           C
         5           1      C      COMPLEX VARIABLE DECLARATION  AND COMPLEX STATEMENTS             C
         6           1      C*************************************************************C


INTERMEDIATE SYMBOL STRING          TEMPORARY SYMBOL TABLE
LENGTH =    400                     LENGTH =    300          OVERFLOW LENGTH =    100
LAST ENTRY =      9                 LAST ENTRY =      5                LAST ENTRY =      0
CURRENT POINTER =      1            CURRENT POINTER =      1           CURRENT POINTER =      0
ZERO EQUAL SIGN =       0

INDEX  CONTENTS              INDEX  CONTENTS   WORDS  OVERFLOW CONTENTS

    1  V                  **     1  COMP LEXF
    2  V                         2  UNCT IONC
    3  V                         3  HECK 1
    4  (
    5  V                         4  ZINP UT
    6  ,
    7  V                         5  MULT
    8  )
    9  $$
```

```
CHECK1 = (ZINPUT + (34.78, 29.6))*MULT - (4.12E-2, 6.5E+3)

INTERMEDIATE SYMBOL STRING
LENGTH =   400
LAST ENTRY =    12
CURRENT POINTER =     1
ZERO EQUAL SIGN =     2

TEMPORARY SYMBOL TABLE
LENGTH =   300                    OVERFLOW LENGTH =   100
LAST ENTRY =     5                LAST ENTRY =     9
CURRENT POINTER =     1           CURRENT POINTER =     9

 INDEX  CONTENTS          INDEX  CONTENTS   WORDS  OVERFLOW CONTENTS
**    1  V              **    1  CHEC K1
      2  =                    2  ZINP UT
      3  (
      4  V                    3     0    1      3  (34. 78,2 9.6)
      5  +
      6  C                    4  MULT
      7  )
      8  *                    5     0    5      4  (4.1 2E-2 ,6.5 E+3)
      9  V
     10  -
     11  C
     12  $$
```

PATH (500) - (The Path Stack)

ALTEDG (2,500) - (The Alternate Edge Stack)

PATH Entry

Node

ALTEDG Entry

Node in Path Stack	Successor

The PATH Stack and the Alternate Edge Stack are two temporary stacks used in tracing the flow of control through a subroutine, an external function, or a main program. The top of the PATH Stack contains the node (the statement number) that is currently being examined.

The PATH Stack entry consists of one (1) computer word.

Each ALTEDG Stack entry consists of two (2) columns, each one (1) computer word wide.

Entry Contents - Words)

Path Stack

1) Node - (Positive Integer)

Node contains the statement number of the node being investigated in the flow of control.

Alternate Edge Stack

1) Node in PATH Stack - (Positive Integer)

This element contains the statement number of a node which exists in the PATH Stack.

2) Successor - (Positive Integer)

This element contains the identification of a successor to the node specified in the Node in PATH Stack.

Coding

Path Stack

Node

Value	Meaning
N	Statement number N is a node in the path being traced.

Alternate Edge Stack

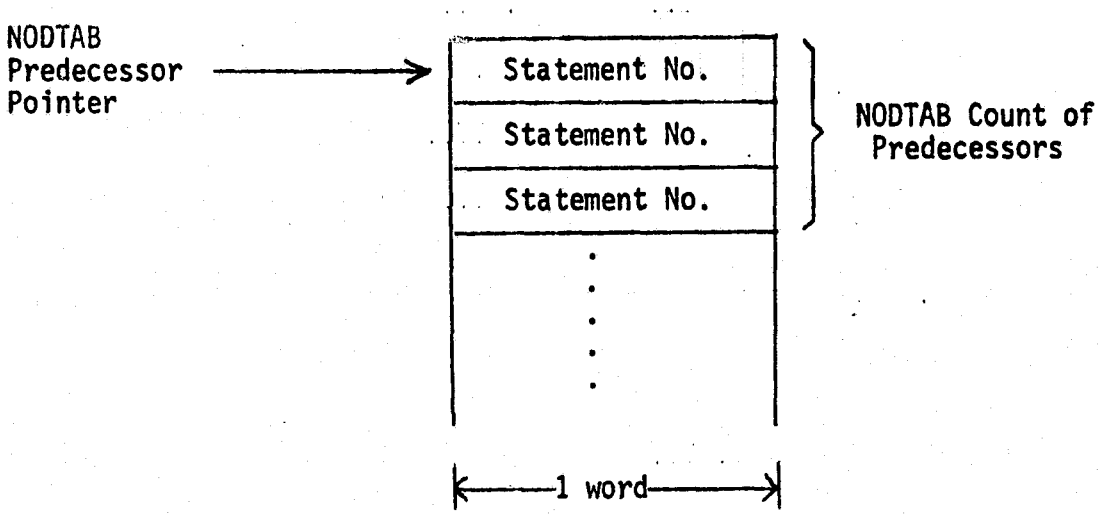
Node in PATH Stack

Value	Meaning
N	Statement number N is a node in the path being traced, and it has at least one successor.

Successor (as in the Successor Codes in the Successor Table)

Value	Meaning
M<10,000	Statement number M is a successor of statement number N.
10,000	The successor contains an external procedure reference.
20,000	The successor is a RETURN statement.
30,000	The successor is an END statement.
40,000	The successor represents a 'Program Halt'
50,000	The successor contains a statement function reference.
60,000	The successor branches through a variable.
90,000	The successor transfers to an undefined label.

PRETAB (1000) - (Predecessor Table)



The Predecessor Table records immediate predecessor information for each source code statement. The Predecessor Table is a set of sequential lists whose structure is defined by NODTAB predecessor entries.

Each predecessor entry is one (1) computer word wide. Entry to a list of predecessors is obtained from the NODTAB Predecessor Pointer. The predecessor list length is determined by the NODTAB predecessor Count.

Entry Contents - (Non-Negative Integer)

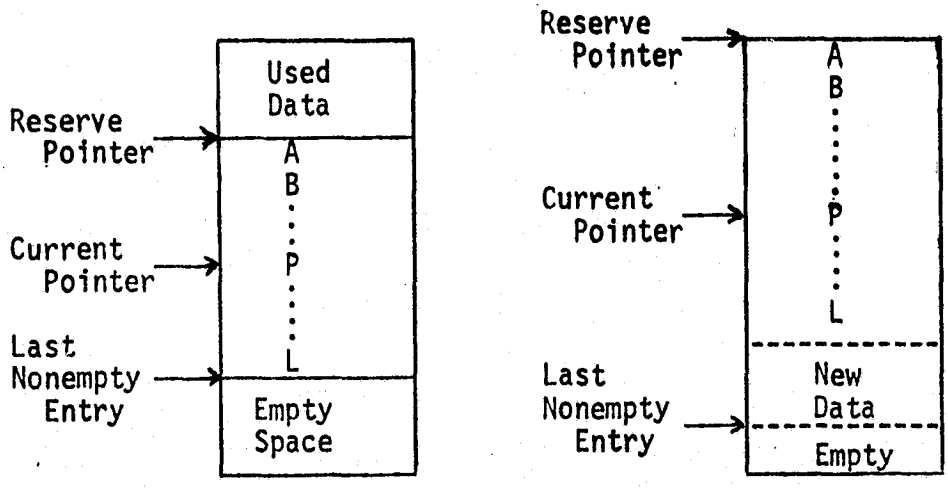
Each Predecessor entry contains either a source statement number (index to a NODTAB position) or a special code indicating program boundary control tansitions.

Coding

PRETAB Entry

Value	Meaning
0	Empty table position
P (≤NODTAB length)	Statement number (index to NODTAB position) of immediate predecessor in flow of control.
70,000	Primary entry point
80,000	Secondary entry point

SCNELM(140) - (Scan Buffer)

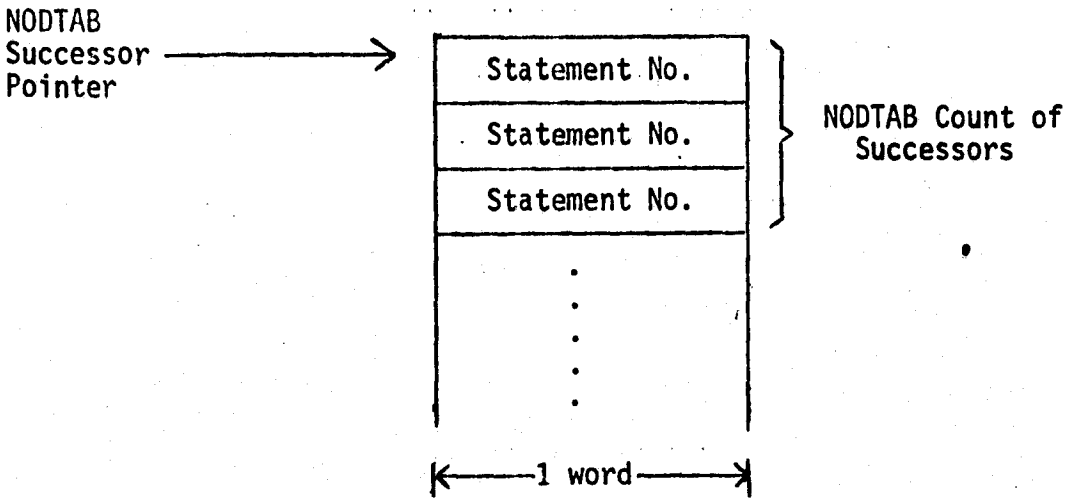


The Scan Buffer normalizes the presentation of FORTRAN statement text to the FORTRAN Front End, combining continuation cards into a linear array. Card image data is placed in the Scan Buffer one card at a time as a string of A1 format characters. For the first card, columns 1 through 5 and columns 7 through 72 are placed in the top of the Scan Buffer. If continuation cards are present, the continuation text in columns 7 through 72 is appended to the end of the buffer.

If more card image data is presented than can fit in the buffer at one time, used data is discarded and the Scan Buffer is compressed by moving elements to the top. The Reserve Pointer marks the deletion boundary of buffer text. As the scanner processes lexical items, the Reserve Pointer is moved down. The Current Pointer marks the next symbol to be delivered to the scanner.

The size of the Scan Buffer controls how many card images can be processed without compression. The buffer size is set to accommodate normal source code and compress on exceptionally long statements.

SUCTAB (1000) - (Successor Table)



The Successor Table records immediate successor information for each source code statement. The Successor Table is a set of sequential lists whose structure is defined by NODTAB successor descriptions.

Each successor entry is one (1) computer word wide. Entry to a list of successors is obtained from the NODTAB Successor Pointer. The length of the successor list is determined by the NODTAB Successor Count.

Entry Contents - (Non-Negative Integer)

Each Successor entry contains either a source statement number (index to a NODTAB position) or a special code indicating program boundary control transitions.

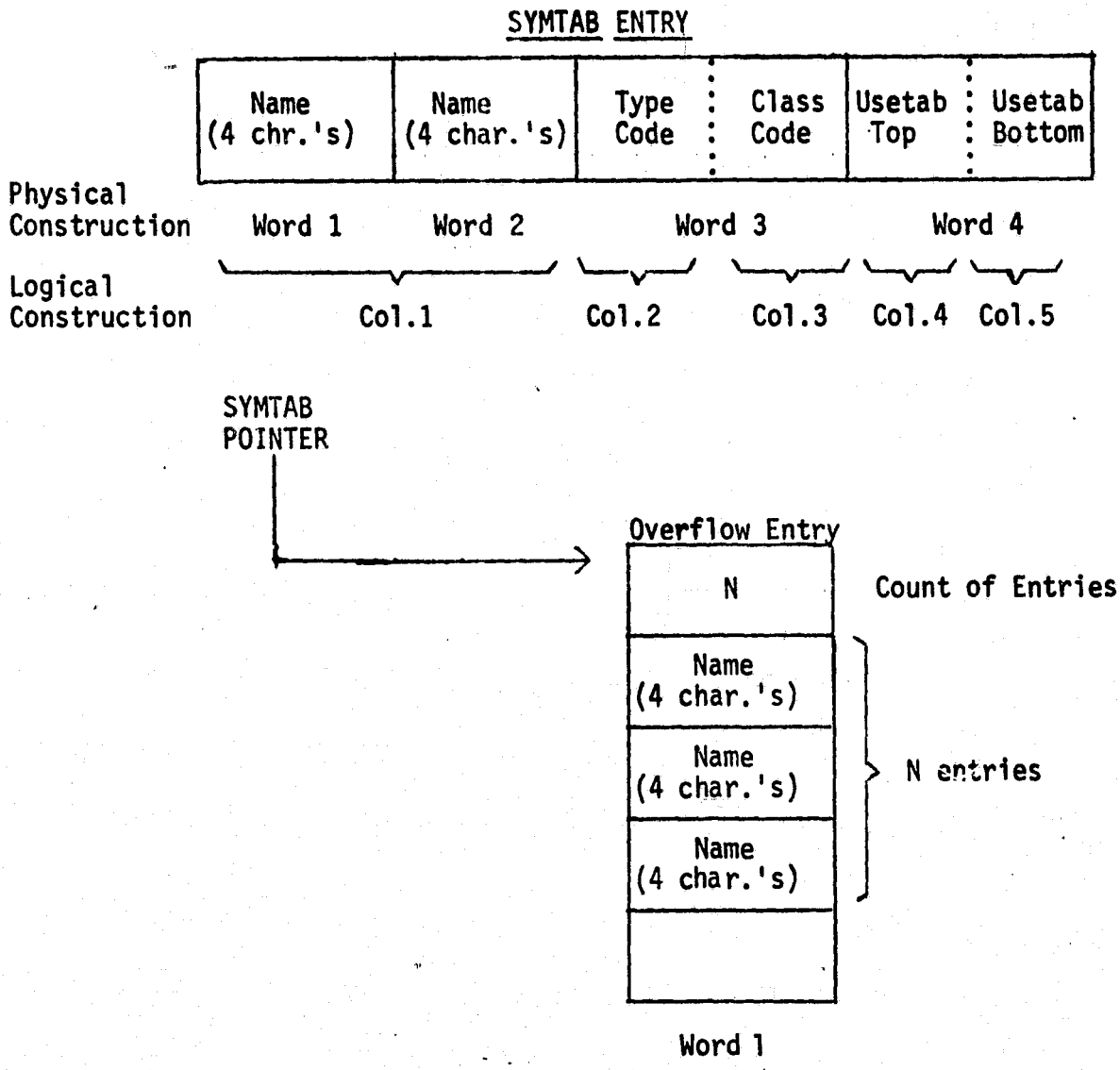
Coding

SUCTAB Entry

Value	Meaning
0	Empty Table position
P (≤NODTAB length)	Statement number (index to NODTAB position) of immediate successor in flow of control.
10,000	External procedure reference
20,000	RETURN
30,000	END statement
40,000	Program halt
50,000	Statement function reference
60,000	Branch through a variable
90,000	Transfer to undefined label

SYMTAB (4,700) - (Symbol Table)

SYMOVR (200) - (Symbol Overflow Table)



The Symbol Table is composed of a main symbol table (SYMTAB) and an overflow table (SYMOVR). Shorter symbolic elements are inserted directly into the main symbol table. Longer symbolic elements are inserted in the overflow table with the main symbol table entry containing a pointer to the overflow entry.

Main Symbol Table entries are hash coded on the first symbolic unit of element information. Overflow entries are sequential based upon the

order of appearance in the source text. Overflow entries are accessed only through pointers in the Main Symbol Table.

Main Symbol Table entries are composed of four (4) computer words organized as five (5) logical columns. Overflow entries are an extension of logical column 1 in the main symbol table.

Entry Contents - (Logical Columns)

Main Symbol Table

1) Symbolic Element - (Alphanumeric Code or Non-Negative Integer)

The character code string composing the symbolic element is provided by this entry element in one of the following ways:

a) Direct - The character string is provided directly in the main symbol table entry. The symbolic element is stored in two computer words, each containing character code information in A4 format. The leftmost word contains the leading characters of the string.

b) Indirect - The main symbol table provides a pointer to the overflow entry where the character string is stored.

2) Type Code - (Non-Negative Integer)

Type code indicates the data type of the symbolic element (see coding).

3) Class Code - (Non-Negative Integer)

Class code indicates the category of name used in the source text (see coding).

4) USETAB Top Pointer - (Non-Negative Integer)

The top pointer is an index to the first use of the symbolic element by the module source code. This pointer provides use table entry to the linked list of element uses in the module. (See USETAB description.)

5) USETAB Bottom Pointer - (Non-Negative Integer)

The bottom pointer is an index to the last use of the symbolic element in module source code.

Overflow Table

1) Count of Entries - (Positive Integer)

The count indicates how many words which follow are symbols of the character string.

2) Symbolic Entry - (Alphanumeric Code)

Symbols of the character string stored in single words. Leading characters are stored in the leftmost position of the top word, progressing left to right, then down the list. Each word contains character data in A4 format.

Coding

Main Symbol Table

1) Symbolic Element

Value	Meaning
0 0	Empty entry in Symbol Table
ALPHA ALPHA	Direct Symbolic Element character data in 2A4 format
0 P	Indirect Symbolic Element data. P is a positive integer pointing to the COUNT entry of the overflow table.

2) Type Codes

Value	Meaning
0	Unknown
1	Floating Point
2	Double Precision
3	Complex
4	Logical
5	Neutral
6	Character Code (Hollerith)
7	Integer

3) Class Code

Value	Meaning
0	Unknown
1	Subroutine Name
2	Statement Function Name
3	Array Variable
4	Function Name
5	Statement Label
6	Scalar Variable
7	Common Block Label
8	Constant
9	Entry Point Name
10	
11	
12	Program Name
13	Temporary Variable
14	
15	Statement Function Dummy Parameter
16	Explicit External Function or Subroutine

4) USETAB Top Pointer

Value	Meaning
0	No uses because of Use Table overflow.
P	Pointer to first use of symbol in USETAB.

5) USETAB Bottom Pointer

Value	Meaning
0	No uses because of Use Table overflow.
P	Pointer to last use of symbol in USETAB.

Overflow Table Entries

Value	Meaning
N	The next N entries contain word units of character code in A4 format.
ALPHA	Character code of symbolic element in A4 format.

Hash Entry to Symbol Table

Entry to the Symbol Table is obtained by considering the integer value of the first 8 characters of the symbol. The first 4 characters and second 4 characters are used as follows to compute the initial entry

VAL = ((first 4 chars.) + (last 4 chars.)) /2

Entry Point = (|VAL| taken Module PRIME) + 1

where,

PRIME is the largest prime number smaller than the Symbol Table size.

(NOTE: On machines which permit more than 4 characters to be stored in an integer word, the leading 4 character strings are right justified with zero left fill rather than taking the absolute value.)

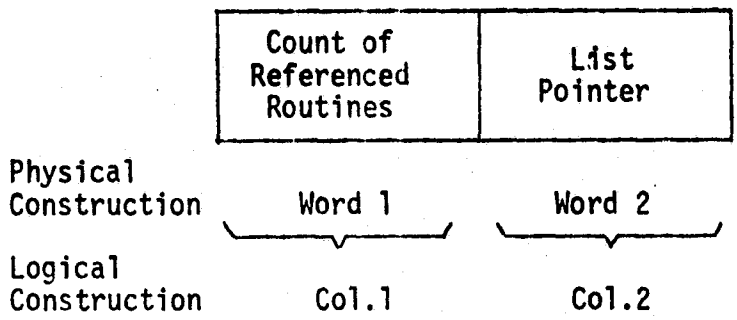
If the initial table entry point is occupied, the table is searched in wrap around fashion from the initial entry until a vacant location is found or the whole table has been searched.

Remarks

The ability to differentiate between Direct Data and Indirect Pointers in the Main Symbol Table Symbolic Elements relies upon left justified character data. If the entry contains left justified character code, the value of the first word will be either a negative integer (leading bit of first char. is 1) or a large positive integer (leading bit of first char. is 0). Thus, any negative value or value larger than a leading character with all zeroes, will be direct data. A positive value smaller than the smallest numeric value of a left justified character string will be a pointer.

For current implementation, the symbolic unit covers two computer words. By right justifying the pointer in the symbolic unit, the leading word contains zero, and the second word contains the pointer. Since an all zero field is not a valid left justified, blank right fill structure for any machine known to the developer, this technique should be transportable to any other computer. (Note: The value of character code is not significant for this technique.)

SHTAB (2,200) - (System Hierarchy Table)



The System Hierarchy Table is a sequential table which is basically a horizontal extension of the Directory. Along with the System Hierarchy to Directory Table (SHDTAB), this table records the calling hierarchy of the software system being analyzed.

Each System Hierarchy Table Entry consists of two (2) computer words organized as two (2) logical columns.

Entry Contents - (Logical Columns)

1. Count of Referenced Routines - (Non-Negative Integer)

   This is a count of the number of different subprograms the routine specified in the Directory references. It is the length of the associated list in the System Hierarchy to Directory Table.

2. List Pointer - (Non-Negative Integer)

   The List Pointer points to the first member of a list in the System Hierarchy to Directory Table.

Coding

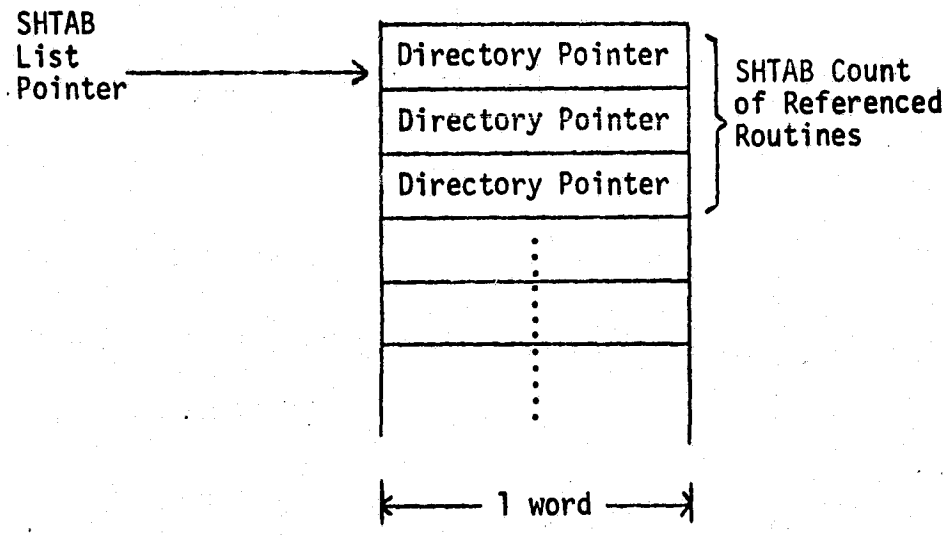
Number of Referenced Routines

Value	Meaning
0	No routines are referenced.
N	N different routines are referenced.

List Pointer

Value	Meaning
0	No routines are referenced.
LP	The list of referenced routines in the System Hierarchy to Directory Table begins in position LP of SHDTAB.

SHDTAB (400) - (System Hierarchy to Directory Table)



The System Hierarchy to Directory Table is a set of sequential lists whosw structure is defined by SHTAB. Along with the System Hierarchy Table, this table records the calling hierarchy of the software system being analyzed.

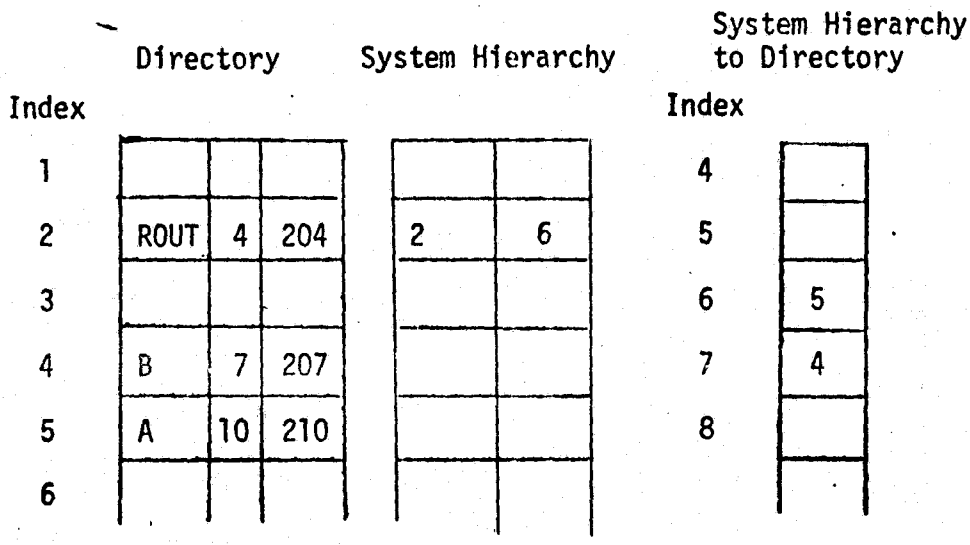
Each System Hierarchy to Directory Table Entry is one (1) computer word wide. Entry into a list of referenced routines is by way of the SHTAB List Pointer. The length of a list is determined by the SHTAB Count of Referenced Routines.

Entry Contents - (Positive Integer)

Each entry points to a module in the Directory.

Example:

```
SUBROUTINE ROUT
      :
Call A
      :
CALL B
```



TRACE (3,400) - (The Trace Stack)

Directory Pointer	Number of Unexamined Successors	Pointer to Next Successor to be Examined

The Trace Stack is a temporary stack used in the tracing of system hierarchy paths. The top of the stack contains information about the location in the hierarchical path that is currently being examined.

Each Trace Stack consists of three (3) computer words organized as three (3) logical columns.

Entry Contents - (Words)

1) Directory Pointer - (Positive Integer)

This element points to a module's identification in the Directory.

2) Number of Unexamined Successors - (Non-Negative Integer)

This element contains the number of different subprograms referenced by the module indicated by the Directory Pointer that have not yet been examined.

3) Pointer to Next Successor to be Examined - (Non-Negative Integer)

This element points to the System Hierarchy to Directory Table (SHDTAB), which in turn points to a module's identification in the Directory. Thus, this element is actually an indirect address.

Coding

Directory Pointer

Value	Meaning
DP	The module's identification is located in row DP of the Directory.

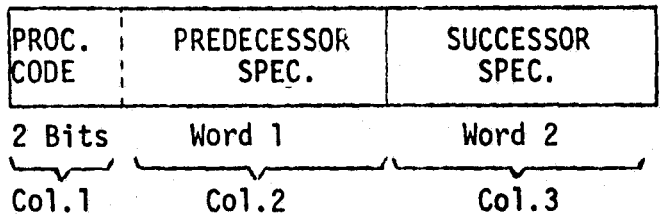
Number of Unexamined Successors

Value	Meaning
0	There are no more successors (referenced routines) to be examined.
N	There are N successors that have not yet been examined.

Pointer to the Next Successor to be Examined

Value	Meaning
0	The module indicated by the Directory Pointer has no successors, i.e., does not reference any subprograms.
N	The location of the module in the Directory is in row N of the System Hierarchy to Directory Table (SHDTAB).

TRIP (2,300) - (Transitions Pairs Table)



The transition pairs table is a sequential list characterizing the branching transitions within a module. Each entry represents a transition other than the normal "next statement" successor to the present statement.

Transitions include the following:

1. Branches within the module.
2. References to external modules (CALL, Function references.
3. References within the module (statement function references).
4. Boundary conditions (entry, return, program halt, etc.)

Entries are made during module processing, marking positions which require reexamination after all statement labels are defined. Postprocessing turns all entries into either node numbers or special codes for unusual transfers.

Each TRIP table entry is composed of two words. The first word contains a flag which is removed by the postprocessing procedure.

Entry Contents

1) Processing Specification - (Two bit Flag)

The processing specification indicates whether either the Predecessor or Successor specification must be replaced by the node number of a label.

2) Predecessor Specification - (Non-Negative Integer)

The predecessor specification indicates the "from" portion of the transition recorded.

3) Sucessor Specification - (Non-Negative Integer)

The Successor Specification indicates the "to" portion of the transition recorded.

Coding

1) Processing Code.

Prior to post-processing the code flag indicates entry positions requiring conversion to node numbers:

0 - No conversion required

1 - Convert successor to node number

2 - Convert predecessor to node number

3 - Convert both to node number.

After postprocessing, the flag is set to 0.

2) Predecessor Specification - (Non-Negative Integer)

0 - Empty entry

Prior to conversion

+N - Symbol table position of label ($N \leq$ symbol table length)

After Conversion

+M - node number of predecessor (M≤node table length)

Special Codes (not converted)

70000 - primary entry point to module

80000 - secondary entry point to module

90000 - predecessor is undefined label.

3) Successor Specification

0 - Empty Entry

Prior to conversion

+N - Symbol table position of label (N≤symbol table length)

After conversion

+M - node number of predecessor (M≤node table length)

Special Codes (not converted)

10000 - Call to external routine

20000 - RETURN statement

30000 - END statement

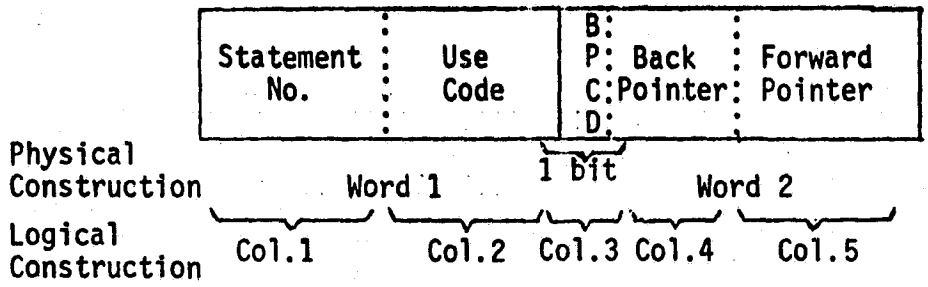
40000 - Program halt

50000 - Statement function reference

60000 - Branch through an assign variable

90000 - Branch to undefined label.

USETAB (2,2000) - (Use Table)



The Use Table is a sequential list with connective pointers to both the Symbol Table and Node Table. The Use Table entries record how symbolic elements of the program are used by source code statements.

Each Use Table entry consists of two (2) computer words organized as four (4) logical columns.

Entry Contents - (Logical Columns)

1) Statement Number - (Non-Negative Integer)

The Statement Number is an index to the Node Table (NODTAB) indicating the source statement to which the recorded use belongs.

2) Use Code - (Non-Negative Integer)

The Use Code indicates how the symbolic element is used in the source code.

3) Back Pointer Code - (One Bit Flag)

The Back Pointer Code indicates whether the pointer is associated with a Symbol Table position or a Use Table Position.

4) Back Pointer - (Flagged Non-Negative Integer)

The Back Pointer forms the first pointer of a double linked list. The Back Pointer indicates either the last usage of the symbolic element in a previous USETAB position, or the index of the symbolic element position in the Symbol Table.

5) Forward Pointer - (Non-Negative Integer)

The Forward Pointer forms the second pointer of a double linked list. The Forward Pointer indicates the next usage of the symbolic element in a succeeding USETAB position.

Coding

Statement Number

Value	Meaning
0	Empty USETAB entry
P	Index to NODTAB for the statement in which this use appears

Use Codes

Value	Meaning
0	Empty Entry
1	Output variable in assignment statement
2	Input to Assignment statement computation
3	I/O Output variable
4	I/O Input variable
5	Do Loop index variable

Value	Meaning
6	Do Loop starting value
7	Do Loop ending value
8	Do loop increment
9	Label definition
10	Transfer to a label
11	Common block entry
12	Data statement variable entry
13	Array declaration
14	Type statement entry
15	Subscript to array
16	Function dummy parameter
17	Subroutine dummy parameter
18	Function actual parameter
19	Subroutine actual parameter
20	Conditional Branch decision variable
21	External procedure reference (e.g., CALL)
22	Variable set in "ASSIGN" statement
23	Transfer through a variable value
24	Index variable used in "COMPUTED GO TO"
25	Do loop termination label
26	I/O unit specification
27	"FORMAT" reference (label or array name)
28	Multiple return point parameter

Value	Meaning
29	Reference to the address of a label
30	BEGIN list bracket
31	END list bracket
32	BEGIN subexpression bracket
33	END subexpression bracket
34	Subexpression output variable
35	Subexpression input variable
36	EQUIVALENCE list member
37	Procedure name in an EXTERNAL statement
38	
39	
40	Declaration
41	DATA value specification
42	Repeat specification
43	Identifying index
44	
45	
46	Statement function "CALL"
47	I/O record specification

Back Pointer

Value		Meaning
BPCD	Pointer	
0	0	No symbol recorded due to Symbol Table overflow or empty entry.
0	P	Pointer value P is an index to the Use Table position in which the symbolic element was last used.
1	P	Pointer value P is an index to the Symbol Table position where the symbolic element is defined.

Forward Pointer

Value	Meaning
0	This is the last use of the symbolic element.
P	The next use of the symbolic element is in position P of the Use Table.

List Brackets. BEGIN/END list bracket codes are used within a single statement to group construction elements. This grouping clarifies the association of members for analysis. Two types of lists are formed:

1. Independant lists. Simple markers surrounding the elements which are associated. Members of the list are not attached to any other program element.
2. Dependent lists. Dependent lists are subordinate to another program element. The list is attached to the referenced element by the back pointer of the BEGIN bracket entry.

Lists are currently used in FACES to set off the following structures:

Independent Lists

1. DATA statement variable lists and constant lists
2. I/O lists enclosed in parenthesis pairs (usually an implied DO construction)
3. Variables associated by an EQUIVALENCE group.
4. Branch lists of Computed and Assigned GO TO statements.

Dependent Lists

1. Actual and Dummy parameter lists of Subroutine, Functions, and Statement Functions. The list is linked to the subprogram name referenced or definition.
2. COMMON variable lists. The list is linked to the COMMON label specified on the statement.

Subexpression Brackets. Subexpression brackets are used like

list brackets to group elements participating in a subexpression. The bracket pair always appears within one statement, delineating elements used to "compute" a temporary variable value.

Special Notes. Most Use codes are self explanatory. In some instances, however, the same code is used for several purposes. The following is an enumeration of the multiple uses:

Use	Meaning	Used For
5, 6, 7, 8	DO variables	DO statements and implied DO loop constructions
15	Subscript	Both array declaration dimensions and subscripts in array references
40	Declaration	Module names on header cards Entry point names Common lables of COMMON blocks
43	Identification	Used on STOP, PAUSE, etc. cards for optional identification

X. COMMON BLOCK DESCRIPTIONS

COMMON/ALI/ALIGN(2,300), PLALI(2)

ALIGN(1,I) - Alignment Table 1. Contains description of model COMMON Blocks or model parameter lists (formal parameter lists) used in COMMON Block Alignment checks or Parameter List Alignment checks respectively.

ALIGN(2,I) - Alignment Table 2. Same as Alignment Table 1, except contains description of comparison COMMON Blocks or comparison parameter lists (actual parameter lists).

PLALI(1) - Pointer to last non-empty (valid) row in Alignment Table 1.

PLALI(2) - Pointer to last non-empty (valid) row in Alignment Table 2.

Physical length of Alignment Tables is stored in COMMON Block /LTEMP/.

For a detailed discussion, see section describing structure of Alignment Tables.

COMMON/ALINFO/NAME(2), MNAME(2,2), SCIND(2), FFIRST(2), LFIRST(2),
FSTAT(2), LSTAT(2), NUMOCC

This COMMON Block contains information necessary for passing COMMON Block and Parameter List alignment vidations to the sort/merge file.

FFIRST, LFIRST, FSTAT, and LSTAT all refer to locations in terms of the relative card number.

a. For Parameter List Alignment, NAME contains the name of the external reference, four characters per word, left adjusted.

For COMMON Block Alignment, NAME contains the name of the COMMON Block, four characters per word, left adjusted.

b. MNAME(1,I) contains the name of the model module (the module the contents of Alignment Table One were derived from), four characters per word, left adjusted.

MNAME(2,I) contains the name of the comparison module (the module the contents of Alignment Table Two were derived from), four characters per word, left adjusted.

c. SCIND(1) contains the source code index of the model module.

SCIND(2) contains the source code index of the comparison module.

d. FFIRST(1) contains the location of the first card of the first statement in the model module.

FFIRST(2) contains the location of the first card of the first statement in the comparison module.

e. LFIRST(1) contains the location of the last card of the first statement in the model module.

LFIRST(2) contains the location of the last card of the first statement in the comparison module.

f. FSTAT(1) contains the location of the first card of the statement being compared in the model module.

FSTAT(2) contains the location of the first card of the statement being compared in the comparison module.

g. LSTAT(1) contains the location of the last card of the statement being compared in the model module.

LSTAT(2) contains the location of the last card of the statement being compared in the comparison module.

h. NUMOCC contains either a running total of the number of occasions violations occurred for a given formal parameter list during the Parameter List Alignment Check, or a running total of the number of occasions violations occurrred for a given COMMON Block during the COMMON Block Alignment Check.

```
COMMON/ALT/ALTEDG(2,500),     PALT
```

ALTEDG - Alternate Edge Stack. Contains alternate paths (alternate edges) to path described in Path Stack.

PALT - Pointer to top of Alternate Edge Stack.

Physical length of Alternate Edge Stack is stored in COMMON Block /LTEMP/.

For detailed discussion, see section describing structure of Path Stack and Alternate Edge Stack.

```
COMMON /ANSI/ ANSIFL, RSANSI, LGANSI, EFANSI, PTANSI
```

ANSI standard name File description

ANSIFL - Unit specification for ANSI File

RSNASI - Record size (not currently used)

LGANSI - Length of file (not currently)

EFANSI - End of file indicator (not currently used)

PTANSI - Pointer to file (not currently used)

COMMON/CDBUFF/CARD(80), LCARD, PCARD, PLCARD

This COMMON block holds the FORTRAN source code from a single card image.

CARD - Vector of characters in A1 format

LCARD - Length of character vector

PCARD - Pointer to current card column

PLCARD - Pointer to last nonempty entry of vector

The card image is empty when PLCARD contains zero value.

COMMON/CDCNT/SCINDX, CURCD, BGNCD, ENDCD

This COMMON block contains Source Code Catalogue pointers used by the FFE while constructing the SCAT file. Used to establish SCAT position for FORTRAN statements.

SCINDX - Source code origin index to the module being processed. Contains a zero origin base balue for SCAT file access to the start of a module. Initial value is determined at the start of FFE operation and advanced as modules are processed.

CURCD - Current card count relative to beginning of the module. This is a module relative count for the card image currently occupying the Source Code Input Card Image Buffer. Contains value zero before any input source cards are read.

BGNCD - Module relative count of the first card for a FORTRAN statement. Set by the Scan Buffer Manager as card image source is transferred to the Scan process.

ENDCD - Module relative count of the last card for a FORTRAN statement. Set by the Scan Buffer Manager as subsequent card images are passed to the Scan process.

```
COMMON/CIMAGE/CMCD(80), LCMCD, PCMCD, PLCMCD
```

This COMMON block holds the command card image for a user command.

CMCD - Vector of characters in A1 format.

LCMCD - Length of character vector.

PCMCD - Pointer to current command card column.

PLCMCD - Pointer to last nonempty entry of command card image text.

The command card image is empty when PLCMCD contains a zero value.

COMMON/CMDITM/CLSFY, CITEM(20), LCITEM, PCITEM, PLCITEM

This COMMON Block contains a command item extracted from the command card image. The command item is characterized by item text and qualifiers.

1) CLSFY is a classification of the command item containing one of the following codes:

$$	- end of command card
$$$$	- end of command set
A	- alphabetic item
N	- numeric item
AN	- alphanumeric item
S	- special symbol

2) CITEM is a vector of command item text.

CLSFY	CITEM Contents
$$	$$
$$$$	$$$$
A	Character of command item in A1 format
N	" " " " " " "
AN	" " " " " " "
S	Single special symbol character.

3) LCITM - length of command item vector

4) PCITM - pointer to command item vector position

5) PLCITM - pointer to last nonempty entry of command item vector entries.

COMMON/CMDSYM/ENDCMD, FINCMD, ALPHUM, ALPH, NUM, SPECL

Read - only table of command item classifications.

		SYMBOL (A4)	MEANING
ENDCMD	-	$$	- End of command card
FINCMD	-	$$$$	- Finish of command card set
ALPNUM	-	AN	- Alphanumeric command item
ALPH	-	A	- Alphabetic command item
NUM	-	N	- Numeric command item
SPECL	-	S	- Special symbol command item.

Used to establish command item classification for processing control commands.

```
COMMON/COM/COMTAB(3,100), LCOM, PCOM, PLCOM
```

COMTAB - COMMON Block Name Table. Contains sequential list of all COMMON Block names which appear in software system being analyzed.

LCOM - Physical length of COMMON Block Name Table.

PCOM - Current row pointer to COMMON Block Name Table.

PLCOM - Pointer to last non-empty (valid) row in COMMON Block Name Table.

For a detailed discussion, see section describing structure of COMMON Block Reference Tables.

COMMON/CONFIG/VER, MODLVL, HOST(2), FORTRG(2)

Description of current configuration.

VER - Version of FACES current operating (integer value).

MODLVL - Modification level of current system (integer value).

HOST - Character string of the host equipment for which the system is adjusted. (Alphanumeric value)

FORTRG - FORTRAN target machine for which system is adapted. (Alphanumeric value)

Used for configuration control. Included in global header to avoid old tables being provided to incompatible future version.

Used to print heading on output at start of run.

COMMON /CTRL/ CTRLFL, RSCTRL, LGCTRL, EFCTRL, PT TRL

Control card file description

CTRLFL - Unit specification for Control File

RSCTRL - Record Size for Control File Records (not currently used)

LGCTRL - Length of control File in records (not currently used)

EFCTRL - End of file indicator

PTCTRL - Pointer to control file records (not currently used)

COMMON/CURMOD/MODNAM(2), MODTYP, MODNUM, SCORIG, SCEND

Description of current module being processed. Used in FFE to control table generation and in Report Generator to control primary listings.

MODNAM - Symbolic name of module being processed in 2A4 format. Set to zero prior to establishing name.

MODTYP - Module type code characterizing module. (See DIRECTORY description for coding).

MODNUM - Module number established to connect module with analysis tables for module.

SCORIG - Module source code origin in Source Code Catalogue. Zero origin base address to first card of module.

SCEND - Source code end. Number of card images belonging to module.

COMMON/CURSTM/STNO, STMTYP, FSTUSE, BCD, ECD

Description of current statement being processed. Partial image of NODE table entries to be inserted.

STNO - Statement number assigned to FORTRAN statements within a module.

STMTYP - Statement type code (See NODE table description for values).

FSTUSE - First USE table position for USEs by statement components. Contains value of zero if USEs not recorded for statement.

BCD - Module relative card number of first card on which statement text appears.

ECD - Module relative card number of last card on which statement text appears.

```
COMMON/DIR/DIREC(4,200), LDIR, PDIR, PLDIR
```

DIREC - Module directory. Contains names of modules defined and referenced by software system under analysis. Each module is characterized by type. Each entry contains access information to module local tables and source code card images. (See data structure description for DIRECTORY).

LDIR - Physical length of directory

PDIR - Current row pointer to directory entry

PLDIR - Last nonempty row pointer to valid entries

```
COMMON/DIRCHR/AMTYP(6), LAMTYP, NOVAL
```

Character string decode of module type codes for Directory entries.

AMTYP - Table of character strings in A4 format corresponding to module type codes of 1 through 6.

LAMTYP - Length of decode vector.

NOVAL - Character code in A4 format for invalid module type codes.

```
COMMON/FFENAM/BNAME(2), BLKDATA(2), EMPTY(2)
```

Read-only table of default names used by FFE.

BNAME - Symbolic name assigned to blank COMMON.

BLKDATA - Symbolic name assigned to BLOCK DATA.

EMPTY - Empty name used to position symbol table to empty entry.

COMMON/FFEOPT/PRTSRC, PRTPRS, PRTLTB, PRTDIR, LSTATS

Control variables to initiate maintenance trace options for FFE checkout. Values are set by DATA statement to indicate trace desired.

Values	Meaning
0	No trace
1	Trace action

Variable	Action
PRTSRC	Print source code as it is analyzed by the FFE.
PRTPRS	Print display of Parsing tables after Scan and prior to beginning statement parse.
PRTLTB	Print local tables produced for each module analyzed.
PRTDIR	Print contents of directory after all modules analyzed.
LSTATS	Collect and display statistics on use of local table space.

```
COMMON/FFESYM/ENDSTM, HFUNC, HTION, HIF, HDO, HERR, HEND
```

FORTRAN Front End character string symbols

ENDSTM - $$ - end of statement code

HFUNC, HTION - FUNCTION
HIF - IF
HDO - DO
HERR - ERR
HEND - END

} Character strings used in parsing

```
COMMON /FLAG/ FLAGFL, RSFLAG, LGFLAG, EFFLAG, PTFLAG
```

Flag File description

FLAGFL - Unit specification for file

RSFLAG - Record size for Flag File entries (not currently used)

LGFLAG - Length of Flag File (not currently used)

EFFLAG - end of file indicator

PTFLAG - Pointer to Flag File records (not currently used)

COMMON /FMSG/ FMSGFL, RSFMSG, LGFMSG, EFFMSG, PTFMSG

FORTRAN Message File Description

FMSGFL - unit specification for Fortran Massage File

RSFMSG - record size of Fortran Message records (not currently used)

LGFMSG - Length of file (not currently used)

EFFMSG - End of file indicator for Fortran Message File.

PTFMSG - Pointer to Fortran Message File (not currently used)

COMMON/GLOLNG/LNGGLO(7)

Read-only table of Global table lengths. Allocation of entries is:

LNGGLO(1) - Length of Directory

LNGGLO(2) - Length of System Hierarchy table

LNGGLO(3) - Length of System Hierarchy to Directory table

LNGGLO(4) - Length of Inverse System Hierarchy table

LNGGLO(5) - Length of Inverse System Hierarchy to Directory table

LNGGLO(6) - Length of COMMON name table

LNGGLO(7) - Length of Link List Table of COMMON names to Directory entries.

Used for global header construction and verification. Controls table size change compatibility with existing analysis files.

COMMON BLOCK DESCRIPTION

COMMON/GHD/GHEADR(28), LGHD

GHEADR - Global header for table file

Allocated as follows:

GHEADR(1) - FACES Version producing the tables (INTEGER)

GHEADR(2) - FACES Modification level producing the tables (INTEGER)

GHEADR(3-4) - Host machine on which FACES ran to produce tables (ALPHANUMERIC - 2A4 FORMAT)

GHEADR(5-6) - Target machine FORTRAN for which FACES was adapted. (ALPHANUMERIC - 2A4 FORMAT)

GHEADR(7) - End of SCAT file entries.
Last absolute card image on source code catalog. (INTEGER)

GHEADR(8) - Last module number used for directory entries. (INTEGER)

GLOBAL TABLE SIZES

GHEADR(9) - Directory length (INTEGER)

GHEADR(10) - System Hierarchy table length (INTEGER)

GHEADR(11) - System Hierarchy to Directory List length (INTEGER)

GHEADR(12) - Inverse System Hierarchy table length (INTEGER)

GHEADR(13) - Inverse System Hierarchy to Directory List table length (INTEGER)

GHEADR(14) - Common Block table length (INTEGER)

GHEADR(15) - Link List of Common to Directory length (INTEGER)

LAST ENTRY POINTERS TO GLOBAL TABLES

GHEADR(16) - Last Directory Entry (INTEGER)

GHEADR(17) - Last System Hierarchy Entry (INTEGER)

GHEADR(18) - Last System Hierarchy to Directory List Entry (INTEGER)

GHEADR(19) - Last Inverse System Hierarchy Entry (INTEGER)

GHEADR(20) - Last Inverse System Hierarchy to Directory List Entry (INTEGER)

GHEADR(21) - Last Common Block Table Entry (INTEGER)

GHEADR(22) - Last Entry in Link List of Common to Directory Table (INTEGER)

Local Table Sizes

GHEADR(23) - Symbol Table length (INTEGER)

GHEADR(24) - Symbol Overflow Table length (INTEGER)

GHEADR(25) - Use Table length (INTEGER)

GHEADR(26) - Node Table length (INTEGER)

GHEADR(27) - Successor Table length (INTEGER)

GHEADR(28) - Predecessor Table length (INTEGER)

COMMON/GLO/GLOBNO

This is the Global Number count. Initially set to one, it is incremented by one whenever a violation is to appear in the secondary or display listings and

1) An uninitialized variable is found.
2) A DO loop control used after the loop terminated normally is located.
3) A different formal parameter list is being examined by the Parameter List Alignment Check.
4) A different COMMON Block is being examined by the COMMON Block Alignment Check.
5) A different cyclic calling sequence is found.

```
COMMON /GLOQRY/ GQUERY(20), LGQ
```

Read cnly list of Global query numbers

GQUERY - vector containing global query numbers

LGQ - length of global query vector

COMMON/H/HA, HB, HC, ..., HZ, HO, H1, ..., H9, HEQ, HLP, HRP,
HCO, HPE, HSL, HPL, HMI, HAS, HDOL, HQU, HDQU

Contains character set for system composed of:

HA through HZ - characters A through Z in A1 format

HO through H9 - characters 0 through 9 in A1 format

Special Symbols:

= ( ) , . /
+ - * $ ' " in A1 format

COMMON/INTSS/ISS(400), LISS, PISS, PLISS, LLISS, EFLAG

Intermediate Symbol String of Parsing Tables. Used by FFE to parse statements.

ISS - Intermediate Symbol String vector. Contains Alphanumeric and special symbol code that classify items of FORTRAN text.
(See Parsing Tables Description for values)

LISS - Physical length of Intermediate Symbol String

PISS - Current pointer to Intermediate Symbol being processed

PLISS - Last nonempty entry of ISS for a statement's text

LLIS - Logical end of Intermediate Symbol String. Establishes fill limit for placing elements in ISS from statement text.

EFLAG - First zero level equal sign indicator. Contains zero if no zero level equal sign and position +N of ISS if zero level equal sign is present.

COMMON/IS/ISTAB(2,200), LIS, PIS, PLIS

ISTAB - Inverse System Hierarchy Tables. Along with Inverse System Hierarchy to Directory Table, ISTAB describes called-by hierarchy of software system being analyzed.

LIS - Physical length of Inverse System Hierarchy Table.

PIS - Current row pointer to Inverse System Hierarchy Table.

PLIS - Pointer to last non-empty (valid) row in Inverse System Hierarchy Table.

For detailed discussion, see section describing structure of Inverse System Hierarchy Tables.

COMMON/ISD/ISDTAB(400), LISD, PLISD

ISDTAB - Inverse System Hierarchy to Directory Table. Along with Inverse System Hierarchy Table, ISDTAB describes called-by hierarchy of software system being analyzed.

LISD - Physical length of Inverse System Hierarchy to Directory Table.

PLISD - Pointer to last non-empty (valid) row in Inverse System Hierarchy to Directory Table.

For detailed discussion, see section describing structure of Inverse System Hierarchy Tables.

```
COMMON/JUMPS/TRIP(2,300), LTRIP, PTRIP, PLTRIP
```

Transition Pairs Table used to record internal transitions within a module and references to external modules. (See Transition Pairs Table data description).

TRIP - Recording array for transitions found in a module.

LTRIP - Physical length of recording array.

PTRIP - Current pointer to transition pair entry.

PLTRIP - Pointer to last nonempty entry of transitions recorded.

```
COMMON/LEXITM/ILEX(66), LILEX, PILEX, PLILEX
```

Lexical item constructed by Scanner. Contains characters of lexical item in A1 format.

ILEX - Recording vector for character string of lexical item characters.

LILEX - Physical length of lexical item recording vector.

PILEX - Pointer to current character of lexical item.

PLILEX - Pointer to last nonempty character of lexical item.

```
COMMON/LIN/LINTAB(2,500), LLIN, PLIN, PLLIN
```

LINTAB - Linked List Table. Contains linked lists which indicate in which modules specific COMMON Block declarations appear.

LLIN - Physical length of Linked List Table.

PLIN - Current row pointer to Linked List Table.

PLLIN - Pointer to last non-empty (valid) row in Linked List Table.

For detailed discussion, see section describing structure of COMMON Block Reference Tables.

COMMON/LIS/LISTAB(500), LLIS, PLLIS, MAP(6,20), LMAP, PMAP, PLMAP

LISTAB - List Table. Contains lists used during AIR execution.

LLIS - Physical length of List Table.

PLLIS - Pointer to last non-empty row in List Table.

MAP - List Table Map. Contains descriptions of lists which currently reside in List Table.

LMAP - Physical length of List Table Map.

PMAP - Current row pointer to List Table Map.

PLMAP - Pointer to last non-empty (valid) row in List Table Map.

For detailed discussion, see section describing List Table and List Table Map.

COMMON/LOCLNG/LNGLOC(6)

Read-only table of Local Table Lengths. Allocation of entries is:

LNGLOC(1) - Main Symbol Table length

LNGLOC(2) - Symbol Table Overflow length

LNGLOC(3) - Use Table length

LNGLOC(4) - NODE Table length

LNGLOC(5) - SUCCESSOR Table length

LNGLOC(6) - PREDECESSOR Table length.

Used for global header construction and verification. Controls table sizes used for analysis file creation. Controls compatibility of created tables with future systems.

```
COMMON /LOCQRY/ LQUERY(11), LLQ, SUBLOC
```

Read only table of Local query numbers

LQUERY - vector containing local query numbers

LLQ - length of local query vector

SUBLOC - pointer to last local query entry to be inserted for LOCAL or ALL specification. Queries in positions below SUBLOC are selected only through the ONLY request.

```
COMMON/LSTSTK/LSTSTK(10), LIST, PLST, PLLST
```

Begin/End list bracket Use code stack. Push down stack for connecting Begin/End Use codes within a statement.

LSTSTK - Recording vector for USE Table positions containing Begin Bracket Use codes.

LLST - Physical length of stack

PLST - Current pointer to stack

PLLST - Pointer to top stack entry.

COMMON/LTEMP/LALI, LPATH, LALT, LTRA

This COMMON Block contains the length of the temporary data structures. Each element of this COMMON Block is set in BLOCK DATA and is read-only data.

For most data structures in FACES, the length of the data structure is stored in the same COMMON Block which contains the data structure. But these temporary data structures were designed to exist in main memory at different times, overlaying one another, in an attempt to conserve space. During the overlay process, the information stored in the overlaid data structure simply disappears. Thus, their length must be stored elsewhere.

a. LALI contains the physical length of the Alignment Table.

b. LPATH contains the physical length of the Path Stack.

c. LALT contains the physical length of the Alternate Edge Stack.

d. LTRA contains the physical length of the Trace Stack.

COMMON/MHD/MHEAD(6), LMHD

This COMMON block contains the local modules header data for local tables of a module.

MHEADR(1) - Length of SYMBOL table

MHEADR(2) - Last nonempty entry of SYMBOL OVERFLOW table

MHEADR(3) - Last nonempty entry of USE table

MHEADR(4) - Last nonempty entry of NODE table

MHEADR(5) - Last nonempty entry of SUCCESSOR table

MHEADR(6) - Last nonempty entry of PREDECESSOR table.

COMMON/MISC/HSTARS, HSUB, HEXPR

This COMMON Block contains those literal strings needed during the execution of AIR which do not belong to any other category. They are grouped under the heading 'miscellaneous'. The are set in BLOCK DATA and are read-only data.

a. HSTARS contains the literal string '****'.

b. HSUB contains the literal string '*SUB'.

c. HEXPR contains the literal string 'EXPR'.

COMMON/MSGVEC/DONE(4), LDONE

FORTRAN Front End message vector to suppress multiple table overflow messages.

DONE - Vector of overflow messages issued for a module.

0 - no message issued

1 - message already issued

DONE(1) - NODE table overflow

DONE(2) - PREDECESSOR table overflow

DONE(3) - SUCCESSOR table overflow

DONE(4) - USE table overflow

LDONE - Length of the message recording vector

```
COMMON/MTEXT/KEY, SCATOG, CARD1, CARDN, FLGNUM, FLGCHR(2), OCCUR,
      IORDER, ITEXT(20), ATEXT(2,20), LTEXT, PTEXT, PLTEXT, LASTPL
```

Flag message for attachment to source code of a report. Consists of one or more flags of message text.

KEY - Global sort key for message.

SCATOG - Source code catalogue origin of module text to which message is attached.

CARD1 - First relative card number of source code statement to which message is attached.

CARDN - Last relative card number for source code sequence to which message is attached.

FLGNUM - Flag number for message (integer value).

FLGCHR - Flag symbolic characters for message (Alphanumeric).

OCCUR - Occurrance number from first flag in message.

IORDER - Internal order number of first flag in message.

ITEXT - Integer message text.

ATEXT - Alphanumeric message text.

LTEXT - Physical length of text arrays.

PTEXT - Pointer to text entries.

PLTEXT - Pointer to last nonempty text row.

LASTPL - Recording variable for length of last message delivered. Used to suppress redundant messages.

```
COMMON/NOD/NODTAB(4,700), LNOD, PNOD, PLNOD
```

NODE Table in Local Tables. (See Local Table Data description for contents).

NODTAB - NODE Table array for recording statements of the source modules.

LNOD - Length of NODE Table array.

PNOD - Pointer to current NODE entry.

PLNOD - Pointer to last nonempty entry of NODE contents.

COMMON/NULL/NULL

This COMMON block contains a single throw-away variable used only to excite Functions which position search pointers. Values placed in the variable are never used.

COMMON/ORIENT/HBAK, HFOL

This COMMON Block contains those literal strings needed during the execution of AIR which involve the direction, i.e., the orientation, of a traversal of paths. They are set in BLOCK DATA and are read-only data.

Each element in this COMMON Block contains its own name as a character string, less the leading 'H', left-adjusted.

a. HBAK contains the literal string 'BAK', which represents 'Backtrack'.

b. HFOL contains the literal string 'FOL', which represents 'Follow'.

```
COMMON/PAT/PATH(500),      PPATH
```

PATH - Path Stack. Contains intra-modular flow of control path currently being examined by AIR.

PPATH - Pointer to top of Path Stack.

Physical length of Path Stack is stored in COMMON Block /LTEMP/.

For detailed discussion, see section describing structure of Path Stack and Alternate Edge Stack.

```
COMMON/PERSYM/VVAL(16), SSCODE(16), LVVAL, PVVAL
```

Read-only table of character string templates which are significant if found between a period pair. Used to detect logical constants, relational operators, and logical operators.

VVAL - Contains symbolic character strings in A4 format. Character strings are the first 4 characters of the form to be recognized.

SSCODE - Contains symbolic Intermediate Symbol String code for entry corresponding to the template matched.

LVVAL - Length of template table and ISS code vector.

PVVAL - Pointer to current entry of table during search.

```
COMMON/PRE/PRETAB(1000), LPRE, PPRE, PLPRE
```

PREDECESSOR Table of Local Tables. (See Local Table Data description for contents.)

PRETAB - PREDECESSOR Table for recording statement predecessors of module statements.

LPRE - Length of PREDECESSOR Table.

PPRE - Current pointer entry.

PLPRE - Pointer to last nonempty entry.

```
COMMON /PRNT/ PRNTFL, RSPRNT, LGPRNT, EFPRNT, PTPRNT
```

Print File Description

PRNTFL - unit specification for Print file

RSPRNT - record size of Print File records (not currently used)

LGPRNT - Length of Print File (not currently used)

EFPRNT - End of File indicator (not currently used)

PTPRNT - Pointer to Print File record (not currently used)

COMMON/REDSTK/CCOD(5), VLOC(5), BCOM(5), NSBE(5), RELPRN(5), PRSTK, LRSTK

Comma list reduction stack used by REDCOL to simplify actual parameter lists and array subscript references. (See description of REDCOL for description of reduction stack operation.)

CCOD - Class code of element being processed on current level. (See description of SYMBOL Table for Values.)

VLOC - Position of ISS containing the variable name for a form V( ).

VCOM - Position of ISS where parameter or subscript began. Advanced after each subscript or parameter processed.

NSBE - Flag indicating whether the parameter or subscript needs to be replaced by a subexpression to a temporary.

RELPRN - Relative parenthesis count used to distinguish organizing parentheses of expressions from the closing parenthesis of the parameter or subscript list.

PRSTK - Pointer to top of reduction stack.

LRSTk - Physical length of reduction stack. Established nesting of V( ) forms allowd.

```
COMMON /RESW/ RESWFL, RSRESW, LGRESW, EFRESW, PTRESW
```

Reserved Word File description

RESWFL - unit specification for file

RSRESW - record Size of file records (not currently used)

LGRESW - length of file (not currently used)

EFRESW - end of file indicator (not currently used)

PTRESW - pointer to file records (not currently used)

```
COMMON/SBESTK/SBESTK/SBESTK(10), LSBE, PSBE, PLSBE
```

Subexpression push down stack for recording Begin/End subexpression Use code within a statement. Used to create nested subexpressions.

SBESTK - Recording vector for USE Table positions containing Begin Subexpression Use codes.

LSBE - Physical length of subexpression vector.

PSBE - Current row pointer to stack entry.

PLSBE - Pointer to top of stack.

COMMON/SCALAR/HPLE, HPRM

This COMMON Block contains the literal strings needed during the execution of AIR when referencing the scalar descriptors of a data structure. The elements of this COMMON Block are set in BLOCK DATA and are read-only data.

Each element in this COMMON Block contains its own name as a character string, less the leading 'H', left-adjusted.

a. HPLE contains the literal string 'PLE', which refers to the pointer to the last non-empty row of a data structure.

b. HPRM contains the literal string 'PRM' which refers to the prime number associated with a hash-coded table.

COMMON/SCANBF/SCNELM(140), LSCNE, PSCNE, PLSCNE, RSCNE

Scan buffer for FORTRAN statement text. (See Data Description of SCAN Buffer for contents.)

SCNELM - Scan buffer containing FORTRAN statement text.

LSCNE - Physical length of Scan Buffer.

PSCNE - Pointer to current element of text.

PLSCNE - Pointer to last entry of text contained in buffer.

RSCNE - Reserve pointer to text which has not been completely processed.

COMMON /SCAT/ SCATFL, RSSCAT, LGSCAT, EFS AT, PTS AT

Source code catalogue file description

SCATFL - Source Code Catalogue File unit specification

RSSCAT - Record Size of Source Code Catalogue records

LGSCAT - Length of Source Code Catalogue in number of records

EFSCAT - End of file indicator for Source Code Catalogue

PTSCAT - Pointer to Source Code Catalogue records. (i.e. Associated variable). Points to next record to be read or written.

COMMON/SH/SHTAB(2,200), LSH, PSH, PLSH

SHTAB - System Hierarchy Table. Along with System Hierarchy to Directory Table, SHTAB describes calling hierarchy of software system being analyzed.

LSH - Physical length of System Hierarchy Table.

PSH - Current row pointer to System Hierarchy Table.

PLSH - Pointer to last non-empty row in System Hierarchy Table.

For detailed discussion, see section describing structure of System Hierarchy Tables.

COMMON/SHD/SHDTAB(400), LSHD, PLSHD

SHDTAB - System Hierarchy to Directory Table. Along with System Hierarchy Table, SHDTAB describes calling hierarchy of software system being analyzed.

LSHD - Physical length of System Hierarchy to Directory Table.

PLSHD - Pointer to last non-empty (valid) row in System Hierarchy to Directory Table.

For detailed discussion, see section describing structure of System Hierarchy Tables.

COMMON/STACK/MSTR(20), TSTR(20), LSTR(20), PSTR(20), LSTACK, PSTACK

This is the Control Stack. It consists of four stacks.

MSTR - Module Number Stack Register. Contains module numbers of modules currently being examined by AIR.

TSTR - Table Name Stack Register. Contains names of tables currently being examined by AIR.

LSTR - List Indicator Stack Register. Contains list indicators for tables currently being examined by AIR.

PSTR - Pointer Stack Register. Contains pointers to table locations currently being examined by AIR.

LSTACK - Physical length of Control Stack.

PSTACK - Pointer to top of Control Stack.

For detailed discussion, see section describing structure of Control Stack.

COMMON/SPEREG/ER(10), ERW, IR(10), IRW, TR, LR, PR, CR, FBR, MR

This COMMON Block contains special purpose registers.

a. ER is the Element Register. It contains an element taken from an AIR accessible table which is placed into ER by the GETE (Get Element) subroutine.

b. ERW contains the width, in computer words, of the Element Register.

c. IR is the Immediate Register. It contains an element taken from a list in the List Table which is placed into IR by the LIRL (Load Immediate Register from the List Table) subroutine.

d. IRW contains the width, in computer words, of the Immediate Register.

e. TR is the Table Name Register. It contains the name of the table that is about to be added to the Control Stack by the PUSH subroutine or has just been removed from the Control Stack by the POP subroutine.

f. LR is the List Indicator Register. It contains the list indicator for a list in the table that is about to be added to the Control Stack by the PUSH subroutine or has just been removed from the Control Stack by the POP subroutine.

g. PR is the Pointer Register. It contains the element's location in a table that is about to be added to the Control Stack by the PUSH subroutine or has just been removed from the Control Stack by the POP subroutine.

TR, LR, and PR, are all simultaneously manipulated by the POP subroutine.

h. CR is the Column Number Register. It contains the number of the column most recently accessed by the GETE (Get Element) subroutine. It is only used in debugging.

i. FBR is the Forward-Backward Register. Originally, it was designed to reflect the direction of the flow of control through AIR routines. Currently, there are some exceptions to this definition. For a full explanation, see the sections entitled, "AIR Basic Search Technique" and "Traversing Lists".

j. MR is the Module Number Register. It contains the module number of the module currently residing in main memory.

The Element Register and the Immediate Register have a specific structure. The first element of the array contains the type of information contained in the register, alphanumeric or integer. The second element contains the length, in computer words, of the information contained in the register. The reamining elements contain the actual information placed in the register, starting from the third element.

Example: The character string 'TOGGLE'.

A	2	TOGG	LE						

```
COMMON/SUC/SUCTAB(1000), LSUC, PSUC, PLSUC
```

SUCCESSOR Table in Local Tables. (See Local Tables Data description of contents.)

SUCTAB - SUCCESSOR Table for recording statement successors of module statements.

LSUC - Physical length of SUCCESSOR Table.

PSUC - Current row pointer to SUCCESSOR entry.

PLSUC - Pointer to last nonempty entry of SUCCESSOR Table.

```
COMMON/SYM/SYMTAB(4,700), LSYM, PSYM, SYMPRM, SYMOVR(200), LSYMO,
        PSYMO, PLSYMO
```

SYMBOL Table in Local Tables (See Local Tables Data description for contents.)

SYMTAB - Main Symbol Table.

LSYM - Physical length of Main Symbol Table.

PSYM - Pointer to current entry of Main Symbol Table.

SYMPRM - Largest prime number smaller than physical length of Main Symbol Table. Used in computing hash entry point into Symbol Table.

SYMOVR - Symbol Overflow Table. Contains entries too large to fit into Main Symbol Table data area.

PSYMO - Pointer to current entry of Symbol Overflow Table.

PLSYMO - Pointer to last nonempty entry of Symbol Overflow Table.

LSYMO - Physical length of Symbol Overflow Table.

```
COMMON /SYSQRY/ SQUERY(3), LSQ
```

Read only table of System queries

SQUERY - vector of system query numbers

LSQ - length of system query vector

```
COMMON /TABL/ TABLFL, RSTABL, LGTABL, EFTABL, PTTABL, GHDTR,
              DIRTR, SHTR, SHDTR, ISTR, ISDTR, COMTR, LINTR,
              BASELT, SIZELT
```

Table File descriptor.

TABLFL - Unit specification for Analysis Table file

RSTABL - Record size of Table File data

LGTABL - Length of Table File in number of records

EFTABL - End of file indicator for Table File

PTTABL - Associated variable for pointing to Table File records. Points to next record to be read or written.

GHDTR - Global Header Table File first record

DIRTR - Directory Table File first record

SHTR - System Hierarchy Table File first record

SHDTR - System Hierarchy to Directory Table File first record

ISTR - Inverse System Hierarchy Table File first record

ISDTR - Inverse System Hierarchy to Directory Table File first record

COMTR - Common Table Table File first record

LINTR - Common Table Link List Table File first record

BASELT - Offset Table File space prior to the start of Local Table records.

SIZELT - Number of records occupied by Local Tables of each module

COMMON/TABLE/HCOM, HDIR, HIS, HISD, HLIN, HLIS, HMAP, HNOD,
HPRE, HSH, HSHD, HSTK, HSUC, HSYM, HUSE1, HUSE2

This COMMON Block contains those literal strings needed during the execution of AIR when referencing specific data structures. The elements of this COMMON Block are set in BLOCK DATA, and are read-only data.

Each element in this COMMON Block contains its own name as a character string, less the leading 'H', left-adjusted.

a. HCOM contains the literal string 'COM', which is the abbreviation for the COMMON Block Name Table.

b. HDIR contains the literal string 'DIR", which is the abbreviation for the Directory.

c. HIS contains the literal string 'IS', which is the abbreviation for the Inverse System Hierarchy Table.

d. HISD contains the literal string 'ISD', which is the abbreviation for the Inverse System Hierarchy to Directory Table.

e. HLIN contains the literal string 'LIN', which is the abbreviation for the Linked List Table.

f. HLIS contains the literal string 'LIS', which is the abbreviation for the List Table.

g. HMAP contains the literal string 'MAP', which is the abbreviation for the List Table Map.

h. HNOD contains the literal string 'NOD', which is the abbreviation for the Node Table.

i. HPRE contains the literal string 'PRE', which is the abbreviation for the Predecessor Table.

j. HSH contains the literal string 'SH', which is the abbreviation for the System Hierarchy Table.

k. HSHD contains the literal string 'SHD', which is the abbreviation for the System Hierarchy to Directory Table.

l. HSTK contains the literal string 'STK', which is the abbreviation for the Stack.

m. HSUC contains the literal string 'SUC', which is the abbreviation for the Successor Table.

n. HSYM contains the literal string 'SYM', which is the abbreviation for the Symbol Table.

o. HUSE1 contains the literal string 'USE1', which is the abbreviation for the Linked List Usage Table.

p. HUSE2 contains the literal string 'USE2', which is the abbreviation for the Statement Number Linked Usage Table.

COMMON/TMATCH/MATCH(51), LMATCH, PMATCH, BIAS

Read-only table of character string templates used to detect FORTRAN key words.

MATCH - Contains symbolic character strings in A4 format and empty entries. Character strings are the leading 4 characters of FORTRAN keywords. Empty entries contain zero values.

LMATCH - Physical Length of the template table.

PMATCH - Pointer to current entry of template table.

BIAS - Integer value used in collision process of hash access.

Data description for FORTRAN Key Word Table Operation.

COMMON/LETTR/LTRS(26), LLTRS, PLTRS

Typing vector for establishing variable type code using first letter of variable.

LTRS - Typing vector for alphabetic letters A through Z. Contains type code to be assigned for variables beginning with associated letter.

LLTRS - Physical length of the typing vector.

PLTRS - Pointer to current vector position.

```
COMMON/TMPNAM/TLEAD(8), LTLEAD, MODIF, MODINC, MODBGN
```

Table of characters and modifiers used to generate temporary names in FFE.

TLEAD - Contains read-only character strings to use as the first 4 characters of a temporary name. Names are selected by the type code of the temporary required.

LTLEAD - Physical length of the leading character template vector.

MODIF - Modifier for unique temporary name generation. Used as the second 4 characters of the temporary name.

MODINC - Modifier increment used to advance the modifier symbol string for next temporary.

MODBGN - Beginning character for first temporary within a module.

```
COMMON/TMPSYM/TSTAB(2,300), LTST, PTST, PLTST, TSTOVR(100),
             LTSTO, PTSTO, PLTSTO
```

Temporary Symbol Table of Parsing Tables. The Temporary Symbol Table replicates the structure of the Symbol Table.

TSTAB - Table array of main Temporary Symbol Table.

LTST - Physical length of main Temporary Symbol Table.

PTST - Current row pointer to Temporary Symbol Table entry.

PLTST - Pointer to last entry of main Temporary Symbol Table.

TSTOVR - Overflow space for Temporary Symbol Table entries too long to fit in main entries.

PTSTO - Pointer to current entry of Temporary Symbol Table.

PLTSTO - Pointer to last nonempty entry of Temporary Symbol Overflow space.

LTSTO - Physical length of Temporary Symbol Overflow space.

COMMON/TPCODE/TYPNAM(16)

This COMMON Block contains alphabetic character strings used during the printing of warning messages concerning type mismatches. Each alphabetic string is stored in two consecutive words, four characters per word, left adjusted.

1. UNDEFINED
2. REAL
3. DBL PREC
4. COMPLEX
5. LOGICAL
6. NEUTRAL
7. Character
8. INTEGER

```
COMMON/TRA/TRACE(3,400),    PTRA
```

TRACE - Trace Stack. Contains inter-modular flow of control path currently being examined by AIR.

PTRA - Pointer to top of Trace Stack.

Physical length of Trace Stack is stored in COMMON Block /LTEMP/.

For detailed discussion, see section describing structure of Trace Stack.

```
COMMON/USE/LUSE, PUSE, PLUSE
COMMON     USETAB(2,2000)
```

USE Table for Local Tables (See Local Table data description for contents).

LUSE - Physical length of USE Table.

PUSE - Current row pointer to USE Table entry.

PLUSE - Last nonempty entry of USE Table.

USETAB- USE Table array. Table space is carried in blank COMMON since this is the largest of the Local Tables.

XI. DETAILED MODULE DESCRIPTIONS

Machine Dependency

Host Machine Dependent Routines:

	Bit Manipulations	Character Codes	Random Disk I/O	Sequential I/O (EOF detection)
AINDX	X	X		
ALPHA		X		
CDCATL			X	
CHFECH	X			
CONVER		X		
CONVRT		X		
DIGIT		X		
ENDOFD				X
FLD	X			
FLGFUL	X			
FLGHLF	X			
FNDDIR		X		
FNDSYM		X		
HASHSY		X		
HIFECH	X			
HISTOR	X			
INCOM			X	
INDIR			X	
INGHD			X	

	Bit Manipulations	Character Codes	Random Disk I/O	Sequential I/O (EOF detection)
INISH			X	
INSH			X	
LOFECH	X			
LOSTOR	X			
MVSCAT			X	
OUTCOM			X	
OUTDIR			X	
OUTGHD			X	
OUTISH			X	
OUTLT			X	
OUTSH			X	
PAKCHR	X			
RDCTRL				X
RDFLAG				X
RDFLGF	X			
RDFLGH	X			
RDSCAT			X	
READLT			X	
SHIFTY	X			

Host Machine Dependent COMMON Blocks:

	Bit Size	Character Codes	Random Disk I/O
/HOSTWD/	X		
/TABL/			X
/TMATCH/		X	
/TMPNAM/	X	X	

Target - FORTRAN Dependent Routines:

CONALC

SNZPRO

INTEGER FUNCTION ACCTYP (CURTYP, NEWTYP)

Mnemonic Origin: Accumulate Type

Classification: FFE parsing service routine

Purpose: Accumulate type code for result of an arithmetic expression.

Operation: ACCTYP develops the type code resulting from the evaluation of an arithmetic expression by cumulative examination of expression operand type code.

Normally, the expression begins with an empty type code (value 0). As each member of the expression is processed, the type code of the member is included in the expression type. The resulting type is the highest type found in the expression.

Normal FORTRAN types included in the type accumulation with hierarchy of:

COMPLEX

DOUBLE PRECISION

REAL

INTEGER

LOGICAL.

In addition, FACES extended type of CHARACTER (Hollerith) is a weak member of an expression (same level as LOGICAL). Any expression member higher than LOGICAL or CHARACTER will irreversibly move the accumulation higher than these types.

Parameters:

CURTYP - Current type code for expression accumulated previously.

NEWTYP - New type code of an expression member to be included in the expression.

ACCTYP - Resulting type code returned through the function name.

SUBROUTINE ADDXFR (SCOL, SPCNOD, OARY, OLNG, OPTR, OVER)

Mnemonic Origin: Add Transfer list to a node.

Classification: FFE graph constructing utility.

Purpose: Attach a list of explicit transfers from/to a given node to the list of successors/predecessors to the node.

Operation: ADDXFR uses a sorted TRIP table and descriptors passed by parameter to insert explicit transfers of control to a list of successors/predecessors being constructed.

By assumption, the order of node processing is the same as the sorted order of the TRIP table. The specified node, if it has explicit transfers, will be on top of the current TRIP table entries.

If the specified node number is less than the top entry of TRIP nodes, no explicit transfers are made in the program from/to specified node. If the specified node is equal to the top TRIP entry node, the explicit transfers follow sequentially in the TRIP table. If the specified node is greater than the top TRIP entry, a processing error has occurred and attempts to resynchronize the process are made.

If explicit transfers are found for a node, the list of transfers is inserted into the list provided by parameters. The insertion process is terminated when the node number of TRIP entries changes, the TRIP entries are exhausted, or the provided vector for inserting entries is filled.

ADDXFR is not actually aware of successor or predecessor properties; it works in terms of TRIP table columns. If the selected column is the predecessor entry of a transition pair, the selected node's transfers will be successors of the selected node. Similarly, selecting

the successor column of the pair will produce predecessors for the selected node.

Parameters: SCOL - Selected column of TRIP to locate node numbers of the transition pair.

SPCNOD - Specified node; the node number for which explicit transitions are required.

OARY - Output array (vector) in which to place explicit transitions found.

OLNG - Output array length.

OPTR - Pointer to output array. Advanced for each explicit transfer inserted.

OVER - Overflow indicator. Set if explicit transfers exceed available output array space.

Cross Reference: Also see description of TRIP table and explanation of Program Graph Construction.

INTEGER FUNCTION AINDX(CHAR)

Mnemonic Origin: Alphabetic Index

Classification: Character code support routine

Purpose: Convert alphabetic characters to an index value

Operation: Upon entry, a character is presented for which a numeric index is required. If the character is an alphabetic character, an index between 1 and 26 is returned. If the character is any smbol other than an alphabetic character, the value 0 is returned.

Routine operation is machine dependent based upon the character code values.

Parameters: CHAR - character presented in A1 format.

AINDX - index value generated is returned through the function name.

SUBROUTINE AIR

Mnemonic Origin: Automatic Interrogation Routine

Classification: AIR Driver

Purpose: Drives AIR subsystem.

Operation: Calls routine which satisfy system and user requests. Also controls disposition of certain global tables (i.e., COM, LIN, IS, ISD, SH, and SHD).

Algorithm: Initialize AIR subsystem. AIR assumes list of system and user requests already resides in List Table. As list in List Table is traversed, binary tree search locates external reference to routine which can satisfy request.

Special Note: Zero in request list represents null request.

If global table requested, bring table into main memory if it exists; else create table and store it in secondary storage.

LOGICAL FUNCTION ALPHA (CHAR)

Mnemonic Origin: Alphabetic character.

Classification: System Utility.

Purpose: Determine whether the presented character is alphabetic.

Operation: The character presented by parameter is examined for alphabetic properties. A .TRUE. value is returned if it is alphabetic; otherwise, .FALSE. is returned.

Evaluation of alphabetic properties requires machine dependent operations on the character codes assigned by the host machine. Since the characters are in A1 format (large magnitude integer values), care is required to avoid comparing positive and negative values; if the compare is performed by subtraction, this could result in overflow.

Parameter: CHAR - Subject character to examine for alphabetic properties in A1 format.

SUBROUTINE AMBSYM (NAME, LNG, TYPE, CLASS)

Mnemonic Origin: Ambigious Symbol

Classification: FFE table recording routine

Purpose: Process ambiguous character strings for Symbol Table insertion.

Operation: Upon entry, a symbol has been discovered for which an idential character string is found in the Symbol Table. The Symbol Table entry differs in class and/or type from the currently presented symbol. The Symbol Table may be the desired symbol or a different symbol which simply has the same character string.

For example, if a variable "A" is found in FORTRAN text and a COMMON block label "/A/" has been declared, the Symbol Table entry for both elements will be A. Clearly, these are two different symbols with the same character string.

If, however, a declaration "INTEGER FUN" appears in the source code followed later by a function reference "FUN(A,B)", the Symbol Table entry will contain an entry for FUN as a scalar variable when the function reference is encountered. In this case, the two references are to the same symbol. The Symbol Table entry should be changed from a scalar variable to a function name.

Typing differences may also cause ambiguities. For example, if the symbol "P" is passed as a subroutine parameter will be default typed. If a later declaration of "INTEGER P" appears, the type of P will be forced to integer, causing a type ambiguity, in the Symbol

Table search. To resolve the problem, the sumbol entry type must be changed.

To resolve the ambiguity, AMBSYM either decides to modify the current Symbol table entry to accommodate the new characteristics or make a new entry for the presented symbol. Since class variations require more analysis, the Class conflict is treated first. If the class can be resolved without insertion, type conflict is resolved for the current entry. If Class resolution requires insertion, the sumbol is simply typed as required for a normal insertion.

Upon entry, the Symbol Table is positioned to an entry which has the same character string but differs in type and class specification. The presented symbol's type and class are copied to local variables to permit changing the specification if resoltuion requires modifying the current Symbol Table entry. Note that more than one symbol may have the same character string in the table. The positioned indicated is simply the first occurance of an identical character string. To properly process the ambiguity, all entries of the table with matching symbol strings must be considered before a decision to insert the current symbol can be made. If the ambiguity can be resolved with an entry, the search can be abandoned.

The upper loop treats class conflicts. The current Symbol Table entry is examined extracting the specified class and type. A processing case is selected based upon the required class specified

by the presented symbol. Each case considers the current class and other factors to determine if the selected Symbol Table entry can be modified to satisfy the requirement. If the current symbol cannot satisfy the requirements, insertion is requested. Otherwise, the current entry description is modified and insertion is not required.

If insertion is required by the processing case, the Symbol Table contents are examined for another candidate position. Candidates are exhausted when a matching entry cannot be found before the next empty Symbol Table position is encountered or the Symbol Table is spanned for the case of a Full table.

Type Ambiguity is resolved either by inserting the symbol with normal typing or modifying the type code of the currently selected position.

Parameters: NAME - Symbolic name presented in a vector of A^4 formatted data.

LNG - Length of symbolic character string expressed as number of vector positions.

TYPE - Type specification. Contains either a type code or the value 0 (probably system error if 0).

Special Notes: The most common ambiguity is an array which is recognized as a scalar variable. Since array names may appear

without explicit subscripts, little effort is made in parsing routines to distinguish arrays from scalars.

This routine is responsible for linking the references to statement function parameters in the statement function expression. Incomming references to scalar variable requests are considered if a statement function is currently in progress.

SUBROUTINE ANSIST

Mneumonic Origin: ANSI Standards Function Names

Classification: AIR Query

Purpose: Searches for ANSI Standards function names used not as ANSI Standards functions. Warning flags may be produced for primary listing.

Operation: Algorithm: See Source Code Listing.

SUBROUTINE ANOMLY (ANOCOD, RNAM1,RNAM2, IVAL, AVAL1, AVAL2)

Mnemonic Origin: Processing Anomaly.

Classification: FFE error reporting.

Purpose: Report the occurrance of an unusual processing condition detected during operation.

Operation: Anomaly reports are processing conditions detected during operation which may indicate processing failures. In general, processing anomalies are recoverable events which have been programmed for; they may indicate bad results, however.

The anomaly is reported to indicate where the problem was detected and what caused the difficulty. Since source code being analyzed is not normally printed during FFE operation, the report must also include the module being processed and the card within the module on which the error occurred.

Since anomalies may be reported by the Scan, Parse, or Postprocess routines, the card number is approximated by the larger of the card numbers established in any of these phases.

Parameters:

ANOCOD - Numerical code indicating the nature of the problem detected.

RNAM1 RNAM2 - Alphabetic name of the routine reporting the anomaly.

IVAL - Numerical value of the suspicious variable if appropriate.

AVAL1 AVAL2 - Either alphabetic name of the variable or alphabetic value detected and truncated variable name.

LOGICAL FUNCTION AOPER (ISSCOD)

Mnemonic Origin: Arithmetic Operator.

Classification: FFE parsing utility.

Purpose: Detect operators of arithmetic expressions.

Operation: AOPER identifies Intermediate Symbol String (ISS) entries which correspond to arithmetic operators during arithmetic expression processing. If the presented symbol is an ISS code for arithmetic operators, .TRUE. is returned; otherwise, .FALSE. is returned.

The logical operations of OR, AND, and NOT are considered arithmetic operators. The arithmetic operator ** is treated as two separate symbols * and * in the system; arithmetic expressions are simply scanned, not actually parsed according to operator precedence rules.

Parameters: ISSCOD - Intermediate Symbol String code to consider as an arithmetic operator.

AOPER - Inspection results are passed through the function name.

Cross Reference: See description of Parsing Tables and expression processing.

SUBROUTINE ASNUSE

Mneumonic Origin: Variables Assigned Values But Never Used

Classification: AIR Query

Purpose: Searches for local variables assigned values but never used.

Operations: Program boundaries are not crossed. Variables appearing in parameter list of subprogram are assumed to be "used" in subprogram. Warning flags may be produced for primary listing only.

Algorithm: See Source Code Listing.

SUBROUTINE ATFILE(FILE, LISTNO, ERRFLG)

Mnemonic Origin: Attach File

Classification: AIR General Purpose Utility

Purpose: Adds the contents of a file to the List Table and List Table Map as a list.

Operations: The contents of file FILE are treated as a list. The first available slot in the list Table Map is located, and the list and its description are placed in the List Table Map and List Table. This list is list number LISTNO. If an error occurs, the error flag ERRFLG is set to a positive integer.

Currently, FILE may refer only to the Reserved Word File or the ANSI Standards Function Names File.

SUBROUTINE BAKSCN

Mnemonic Origin: Back up the Scan pointer.

Classification: FFE scan untility.

Purpose: Correct for overscanning in multiple character lexical items.

Operation: The Scan Buffer pointer is moved back one position to recover from access to a character which did not belong to the current lexical item. This short routine is coded stand-alone to avoid proliferation of the Scan Buffer COMMON block to many places in the Scan section of the FFE.

Cross Reference: See description of Source code Scanning in FFE description.

SUBROUTINE BLDCIT

Mnemonic Origin: Build Command Item.

Classification: Control Driver Command card interpretation.

Purpose: Construct the next Command item from Command card image character information.

Operation: Upon entry, a nonempty Command card should occupy the COMMON card image buffer. This information may have been partially processed on previous calls. The pointer to the command card image indicates the next nonblank character on the card image to use in the construction of Command items.

The classification of the Command item is set empty to safeguard against malfunctions in which no item can be found. The card characters are then extracted sequentially to construct the Command item.

If the initial character is a special symbol, a single character Command item is returned with classification "Special".

If the leading character is alphabetic, the classification of "Alphabetic" is established; if numeric, "Numeric" classification is assigned. In either case, characters are extracted until the next blank character, special symbol, or end of card is found. If the additional characters are not the same as the initial character, the classification is altered to Alphanumeric.

After the Command item has been extracted, the pointer to the first character of the Command item text is set and the pointer to the Command card image advanced to the next nonblank character or the end of the card.

The Command item is returned to the calling routine with the text character length set, Command item pointer established to the first character, and classification set to either "Special", "Alphabetic", "Numeric", or "Alphanumeric".

SUBROUTINE BUFMGR

Mnemonic Origin: Scan Buffer Manager.

Classification: FFE Scan service routine.

Purpose: Provide FORTRAN source code to scan routines from card image data and catalogue source code.

Operation: The Scan Buffer Manager is the principal interpreter of FORTRAN card format. FORTRAN text is extracted from the incoming card images and passed to scanning routines through the Scan Buffer.

Card image data is passed one card at a time to the Scan Buffer. As additional statement text is required, the Buffer Manager is called to supply new card image data. Only FORTRAN statement text is transferred, deleting continuation columns and card id fields.

Upon the first call, the card image will be empty since no cards have been read yet. New card data is requested from the physical I/O routine and the data is placed in the Scan Buffer. After processing that data, the scan requests more data looking for continuation cards. If the next card is a continuation, additional card data is placed in the Scan Buffer. If the next card is not a continuation, an end of statement code is placed in the scan buffer and the card image held for the next "first card" request.

When card data is transferred from the card image to the Scan Buffer, the card image is set empty, enabling the reading of additional cards. If the card data is not transferred, the card image buffer is not reset, disabling additional reads.

The Buffer Manager distinguishes requests for first cards from requests for continuation cards by examining the state of the Scan Buffer. If the Buffer has been set empty externally, the request is interpreted as a first card request. If the Scan Buffer is not empty, the request is construed as a request for continuation cards.

Source Code Cataloguing. As source code card images are retrieved from the Source Code Input file, the card data is passed for recording on the Source Code Catalogue. In addition, the module relative card pointers for individual statements are established during the card reading. Acquiring the first card of a statement causes the initial values of the beginning card and ending card to be established for the statement node. If continuation cards are processed, the ending card pointer is advanced.

Comment Cards. If a card image is found to be a comment card, the card is catalogued but no data is transferred to the Scan Buffer. Rather, the card image is set empty and a new card read.

Buffer Compression. Where a series of continuation cards require more space than directly available in the Scan Buffer, the Buffer is compressed. Used data already processed by the scan routines are removed and the data entries moved up. Space is thus made available for new card data.

Terminal Conditions. When an EOF is detected while reading a new card, no card data is returned. Rather, the end of statement code is inserted in the Scan Buffer. This action is taken whether the card requested is for a first card or a continuation card.

Summary. Upon return to the calling routine, FORTRAN card image data is available from the next card for a statement. If the card is a first card, the card image (possibly empty) is returned. If the card should be a continuation card, either continuation data or an end of statement code is returned.

Cross Reference: See Scan Operation description for FFE and data description of Scan Buffer.

SUBROUTINE CBDIM (PARAM1, PARAM2)

Mneumonic Origin: COMMON Block Dimensionality Mismatch

Classification: AIR Query

Purpose: Searches for corresponding COMMON Block entries not having identical dimensions.

Operations: If PARAM1 equals 420, warning flags may be produced for primary listing. If PARAM2 equals 421, warning flags may be produced for secondary listings.

Search for corresponding COMMON Block entries not having identical dimensions. Define scalers of having zero dimensions. Search through COMMON Block declaration halts after first mismatch found.

Algorithm: See Source Code Listing.

Parameters: PARAM1 - Input

PARAM2 - Input

See Also /ALI/ and "Alignment Tables".

SUBROUTINE CBINDS (PARAM1, PARAM2)

Mneumonic Origin: Common Block Individual Size Mismatch

Classification: AIR Query

Purpose: Searches for corresponding COMMON Block entries not having same size.

Operations: If PARAM1 equals 450, warning flags may be produced for primary listing. If PARAM2 equals 451, warning flags may be produced for secondary listing.

Search for corresponding COMMON Block entries not having identical sizes. Search through COMMON Block declaration halts after first mismatch found.

Algorithm: See Source Code Listing.

Parameters: PARAM1 - Input

PARAM2 - Input

See Also /ALI/ and "Alignment Tables".

SUBROUTINE CBNAME (PARAM1, PARAM2)

Mneumonic Origin: Common Block Name Mismatch

Classification: AIR Query

Purpose: Searches for corresponding COMMON Block entries not having identical names.

Operations: If PARAM1 equals 400, warning flags may be produced for primary listing. If PARAM2 equals 401, warning flags may be proudced for secondary listing.

Search for corresponding COMMON Block declarations not having identical sizes. Search through COMMON Block declaration halts after first mismatch found.

Algorithm: See Source Code Listing.

Parameters: PARAM1 - Input

PARAM2 - Input

See Also /ALI/ and "Alignment Tables".

SUBROUTINE CBNENT (PARAM1, PARAM2)

Mneumonic Origin: COMMON Block Number of Entries Mismatch

Classification: AIR Query

Purpose: Searches for corresponding COMMON Block declarations not having same number of entries.

Operations: If PARAM1 equals 400, warning flags may be produced for primary listing. If PARAM2 equals 401, warning flags may be produced for secondary listing.

Search for corresponding COMMON Block declarations not having same number of entries.

Algorithm: See Source Code Listing

Parameters: PARAM1 - Input

PARAM2 - Input

See also /ALI/ and "Alignment Tables"

SUBROUTINE CBTOTS (PARAM1, PARAM2)

Mneumonic Origin: COMMON Block Total Size Mismatch

Classification: AIR Query

Purpose: Searches for corresponding COMMON Block declarations not having same total size.

Operations: If PARAM1 equals 460, warning flags may be produced for primary listing. If PARAM2 equals 461, warning flags may be produced for secondary listing.

Search for corresponding COMMON Block declarations not having same total size.

Algorithm: See Source Code Listing.

Parameters: PARAM1 - Input

PARAM2 - Input

See also /ALI/ and "Alignment Tables".

SUBROUTINE CBTYPE (PARAM1, PARAM2)

Mneumonic Origin: COMMON Block Type Mismatch

Classification: AIR Query

Purpose: Searches for corresponding COMMON Block entries not having identical types.

Operations: If PARAM1 equal 410, warning flags may be produced for primary listing. If PARAM2 equals 411, warning flags may be produced for secondary listing.

Search for corresponding COMMON Block entries not having identical types.

Algorithm: See Source Code Listing.

Parameters: PARAM1 - Input

PARAM2 - Input

See also /ALI/ and "Alignment Tables".

SUBROUTINE CDCATL (CONTRL)

Mnemonic Origin: Card Catalogue.

Classification: FFE physical I/O routine.

Purpose: Record Source code on the Source Code Catalogue.

Operation: The resident card image of the FORTRAN Source Code card is recorded in a sequential fashion on the source code catalogue.

The source code catalogue is a direct access file used in an index sequential fashion. The card image is written on the file unless file space is exhausted. The write process automatically advances the pointer to the file.

Parameters: The parameter to CDCATL is not actually used to control activities. Only the value of 0 is used in calls to CDCATL.

SUBROUTINE CDPRIN (CONTRL)

Mnemonic Origin: Card Print.

Classification: FFE maintenance routine.

Purpose: Provide card printing capability when maintaining or checking the FFE operation.

Operation: CDPRIN permits the printing of source code images as they are processed by the FFE. The lising includes both the statement number and module relative card number alongside the source code image.

CDPRIN is activated by setting the FFE maintenance variable PRTSRC in COMMON block FFEOPT to the value 1.

The parameter to CDPRIN controls whether the call will result in a source code image print or page restoration print. Page restoration is performed at module boundaries when maintenance printing is active. Card image printing is requested when card images are read and catalogued by the BUFMGR routine.

Parameter: CONTRL - determines whether page restoration or card image print should be performed.

SUBROUTINE CDREAD (ENDFIL)

Mnemonic Origin: Card Read

Classification: FFE physical I/O routine

Purpose: Provide physical input from Source Code Input file.

Operation: CDREAD reads input source code from the Source Code Input file until an end of file is detected. Card images are placed in the COMMON card image buffer and the nonempty pointer set to the end of the card. The current column pointer is set to the first card column.

If a read is requested while the card image is still occupied, no physical read occurs. The nonexhausted card image is returned for processing.

If an end of file is detected while reading a card image, the EOF condition is reported to the calling routine through parameter value and recorded in the File descriptor COMMON block. The card image is set empty. No further reads will be made until the EOF indicator is reset externally; any subsequent requests will return EOF/empty card results.

With each physical read, the current card count is advanced. The current card count is used to establish card to statement correlation and source code catalogue entry values.

Parameter: ENDIL - logical output flag to calling routine that EOF condition was encountered on the read or an EOF condition was pending when the read was requested.

Designer's Comments: The avoidance of reads into nonempty buffers was a protective strategy. This feature is not used in the system and will be removed in future versions.

SUBROUTINE CHGCLS (NECLS)

Mnemonic Origin: Change Class.

Classification: FFE table recording routine.

Purpose: Change the class of a symbol to another class.

Operation: Upon entry, the symbol table is positioned to a symbol for which a change of class is required. The new class code presented by parameter is inserted into the symbol's class code.

Parameters: NEWCLS - class code to substitute for current class code of symbol.

Designer's Note: CHGCLS is an interim processing routine used to accomplish class changes before the symbol ambiguity process was crystallized. Class changes will be moved to the ambiguity process in future versions (see AMBSYM).

SUBROUTINE CLRLTB

Mnemonic Origin: Clear Local Tables.

Classification: FFE process preparation routine.

Purpose: Clear contents of local tables.

Operation: Local tables for a module are cleared and set empty. The table contents are reset to zero values in all postions. Pointers to last entries are set empty (value zero).

Designer's Note: Currently, the full length of all tables are reset to zero value. Future versions may clear only the used areas of tables to increase processing speed.

SUBROUTINE CLRMTB

Mnemonic Origin: Clear Module Tables.

Classification: FFE processing preparation routine.

Purpose: Reset module processing tables used by the FFE to analyze a module.

Operation: Processing vectors and tables unique to the FFE are reset prior to processing a module. Simple recording vectors and tables are reset to zero value. The typing vector for FORTRAN variable names is reset to default typing. Temporary name generation is reset.

INTEGER FUNCTION CHFECH (ARRAY, LNG, CHNUM)

Mnemonic Origin: Character Fetch.

Classification: FFE character manipulating utility.

Purpose: Extract a single character from a packed vector of A4 character data.

Operation: CHFECH is provided with a vector containing characters packed in A4 format. A particular character is desired from this array. The character is extracted left justified with zero right file. Blank right fill is inserted to convert the character to A1 format.

If the character specification is inconsistent with the array presented, an error is reported and the value zero returned.

Parameters: ARRAY - Vector containing data packed in A4 format.

LNG - Length of vector passed containing packed characters.

CHNUM - Character number to extract from packed data.

Special Note: Dimensionality of the input packed data is not significant if the vector passed is of unit length. A scalar word or multidimensional array can be treated by CHFECH provided the length is indicated to be 1 and character number in the bound of 1 to 4.

SUBROUTINE CMDEND

Mnemonic Origin: Command End.

Classification: Control Driver Command Card interpretation.

Purpose: Affirmative acknowledgement that a Command card has been interpreted.

Operation: Upon entry, a Command card has been fully processed by the interpretation routine. The card may contain extraneous information or may be an unrecognizable command.

To flush this information, calls are issued requesting Command items until an end of card classification is found. The end of card is acknowledged by setting the Command item empty.

CMDEND provides a single synchronizing point for Command processing on the variable format Command cards.

SUBROUTINE COMBAL (ARRAY, ARRSIZ)

Mneumonic Origin: Common Block Alignment Check

Classification: AIR Query Driver and AIR Query

Purpose: Drives COMMON Block Alignment Check and searches for COMMON Blocks appearing in only one module.

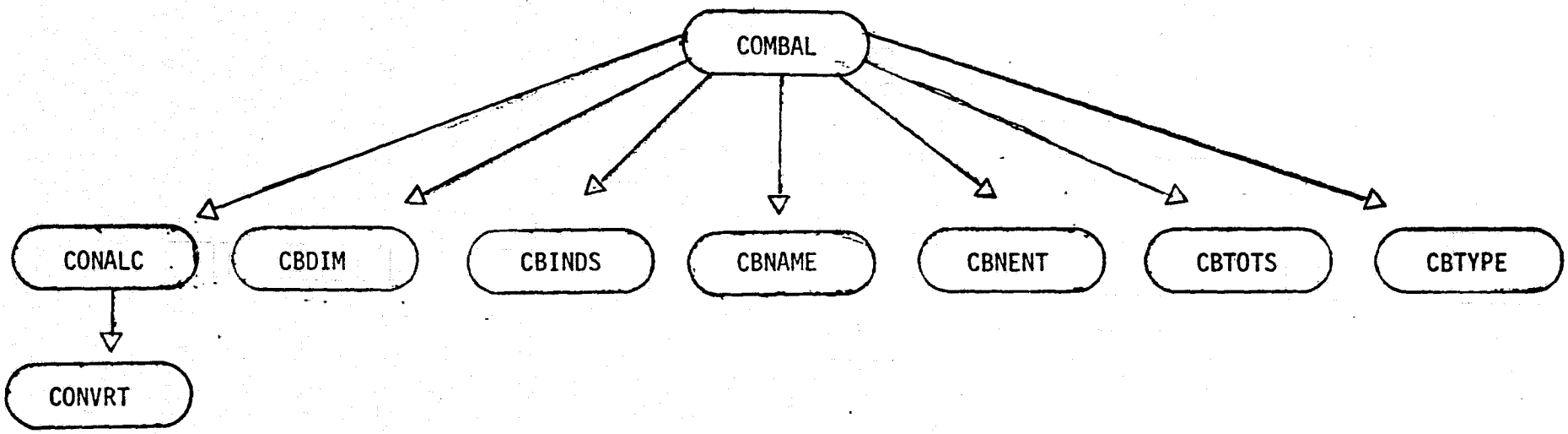
Operations: All Information necessary for COMMON Block Alignment Check is placed in Alignment Tables. Warning flags for COMMON Blocks appearing in only one module may be produced for primary listing only. Array ARRAY has ARRSIZ entries; each entry specifies which COMMON Block Alignment Check is to be performed and whether warning flags may be produced for primary or secondary listings.

Algorithm: See Source Code Listing

Parameters: ARRAY - Input

ARRSIZ - Input

See also /ALI/ and "Alignment Tables".



Common Block Alignment Check,

excluding General Purpose Utilities

SUBROUTINE COMPRS (NEED)

Mnemonic Origin: Compress the Scan Buffer.

Classification: FFE Scan support routine.

Purpose: Provide data space in the Scan Buffer by deleting used character data.

Operation: Upon entry to COMPRS, the Scan Buffer is occupied with card image data from a single statement of FORTRAN text. To complete the statement, additional text must be inserted in the Scan Buffer, but insufficient room remains.

Scan Buffer space is made available by deleting used data from the Scan Buffer and compressing unused data to the top of the array. The reserve pointer on the Scan Buffer indicates the boundary between used und unused data.

If the used data space is insufficient to accommodate the new data, unused data is discarded as an emergency measure and an anomaly is reported. For protection, the smaller value of the reserve pointer and current pointer to the Scan Buffer is used to establish the deletion region.

After the unused data has been adjusted to the top of the Scan Buffer, the pointers are adjusted to point to the same data items they addressed upon entry.

Upon return from COMPRS, the required data space has been created in the Scan Buffer.

Parameters: NEED - Indicates how many Scan Buffer positions are required to accommodate new data.

SUBROUTINE CONALC(TAB, OVFLAG, ERFLAG)

Mnemonic Origin: Construct Alignment Tables for Common Block Alignment Check

Classification: AIR Special Purpose Utility (Referenced only during COMMON Block Alignment Check).

Purpose: Places COMMON Block description in Alignment Table TAB for COMMON Block Alignment Check.

Operations: Place all salient information concerning COMMON Block named in array NAME (see /ALINFO/) in Alignment Table TAB (TAB equals one or two). If the Alignment Table TAB overflows, set overflow flag OVFLAG to one. If unrecoverable error occurs, set error flag ERFLAG to one.

Algorithm: See Source Code Listing.

Parameters: TAB - Input
OVFLAG - Output
ERFLAG - Output

See also "Alignment Tables" and /ALI/.

SUBROUTINE CONALP (PBEGIN, TAB, FSTAT, LSTAT, OVFLAG, ERFLAG)

Mnemonic Origin: Construct Alignment Table for Parameter List Alignment Check

Classification: AIR Special Purpose Utility (referenced only during Parameter List Alignment Check).

Purpose: Places Parameter List description in Alignment Table TAB for Parameter List Alignment Check.

Operations: Place all salient information concerning parameter list indicated by PBEGIN (which points to USE Table) in Alignment Table TAB (TAB equals one or two). FSTAT and LSTAT are set to location of first and last card respectively in which statement containing parameter list resides. If Alignment Table TAB overflows, overflow flag OVFLAG is set to one. If unrecoverable error occurs, error flag ERFLAG is set to one.

Algorithm: See Source Code Listing.

Parameters: TAB - Input
PBEGIN - Input
FSTAT - Output
LSTAT - Output
OVFLAG - Output
ERFLAG - Output

See also "Alignment Tables", /ALINFO/, and /ALI/.

SUBROUTINE CONCOM

Mneumonic Origin: Construct COMMON Block Reference Tables

Classification: AIR Request Routine

Purpose: Construct COMMON Block Name Table and Linked List Table

Operation: Algorithm: See Source Code Listing

Update Global Header to reflect actual nonempty size of each table.

See also "COMMON Block Reference Tables".

SUBROUTINE CONISH

Mneumonic Origin: Construct Inverse System Hierarchy Tables

Classification: AIR System Request Routine

Purpose: Construct Inverse System Hierarchy Table and Inverse Systme Hierarchy to Directory Table.

Operations: Inverse System Hierarchy Tables are derived directly from System Hierarchy Tables. Update Global Header to reflect actual nonempty size of each table.

See also "Inverse System Hierarchy Tables".

SUBROUTINE CONSH

Mneumonic Origin: Construct System Hierarchy Tables

Classification: AIR System Request Routine

Purpose: Construct System Hierarchy Table and System Hierarchy to Directory Table

Operations: Algorithm: See Source Code Listing

Update Global header to reflect actual nonempty size of each table.

See also "System Hierarchy Tables".

INTEGER FUNCTION CONVER (ARRAY, COUNT)

Mnemonic Origin: Convert from alphabetic to numeric value.

Classification: System utility.

Purpose: Convert a decimal integer value represented as a vector of single digits to a positive integer.

Operation: CONVER is passed an array (vector) containing decimal digits in A1 format. This number is to be converted to an integer value.

For protection, leading characters which are not decimal digits, are skipped if they are present in the vector. Remaining digits are converted to a decimal equivalent until either the number of positions have been spanned or a nondecimal character is encountered.

Parameters: ARRAY - one dimensional vector containing decimal digit characters in A1 format.

COUNT - indicator of the number of digits contained in the vector (i.e., the number of array positions to consider in the conversion).

CONVER - the integer value converted is returned through the function name.

FUNCTION CONVRT(WORD1, WORD2)

Mnemonic Origin: Convert Character String into Integer

Classification: AIR Special Purpose Utility

(Referenced only during COMMON Block Alignment check.)

Purpose: Convert character string of numbers into decimal value.

Operations: CONVRT is set to the integer value of the character string stored in WORD1 and WORD2, four characters per word, left-adjusted, blank filled. Non-numeric characters in the character string are ignored.

Algorithm: See Source Code Listing

Parameters: WORD1 - Input

WORD2 - Input

CONVRT - Output

SUBROUTINE CYCALL

Mneumatic Origin: Cyclic Call

Classification: AIR Query

Purpose: Searches for cyclic calling sequences.

Operations: System Hierarchy tables are examined for cyclic calling sequences. Warning flags produced for display listing only.

Algorithm: See Source Code Listing.

See also /TRA/ and "Trace Stack"

SUBROUTINE DATVAR

Mneumonic Origin: DATA Statements containing COMMON Variables

Classification: AIR Query

Purpose: Search for DATA statements not in BLOCK DATA containing COMMON Block variables.

Operations: Warning flags produced for primary listing only.

Algorithm: See Source Code Listing.

SUBROUTINE DEL(LIST)

Mnemonic Origin: Delete List

Classification: AIR General Purpose Utility

Purpose: Delete list and all succeeding lists from List Table and List Table Map.

Operation: Delete list number LIST and all succeeding lists from List Table and List Table Map. This conserves space.

Parameters: LIST INPUT

SUBROUTINE DELISS (FROM, TO)

Mnemonic Origin: Delete entries from Intermediate Symbol String.

Classification: FFE processing utility.

Purpose: Delete indicated entries from the Intermediate Symbol String, compressing remaining entries in the process.

Operation: Upon entry, a span of entries in the Intermediate Symbol Table are to be removed. Parameters indicate the bounds of this deletion. Entries are deleted by moving entries below the deletion region (if any) upward and adjusting pointers to the reduced list.

As error protection, if the current pointer to ISS entries is found in the deletion area, the pointer is moved to the end of the deletion area. If this would be outside the valid ISS entries, the pointer is positioned at the end of the new ISS list.

If the boundary pointers are crossed (i.e., FROM greater than TO), no deletion occurs.

Note that in addition to adjustment to the current ISS pointer and last nonempty pointer, the zero level equal sign pointer is adjusted if necessary.

Parameters: FROM - the first ISS entry to be deleted.
TO - the last ISS entry to be deleted.

SUBROUTINE DELPTB (FROM, TO, SPACE)

Mnemonic Origin: Delete entries from Parsing Tables.

Classification: FFE processing utility

Purpose: Delete entries from parsing tables to make room for replacement entries.

Operation: The parameters passed indicate Parsing Table positions (ISS and TSTAB) entries to be removed in terms of ISS table positions. These entries may be simply removed or replaced with another entry. If replacement is to be performed, the number of vacant table positions to reserve is indicated.

To accomplish the deletion, the boundary elements of ISS and TSTAB are computed. Since Temporary Symbols are associated only with operand entries in ISS, the last TSTAB entry must be examined; if the last ISS entry is an operand, the boundary TSTAB entry is to be deleted; otherwise, the boundary TSTAB entry belongs to another ISS entry and should be kept.

Once the boundaries of deletion have been established, ISS and TSTAB are individually reduced, adjusting pointers for the reduced list. If the current pointers to ISS and TSTAB are found in the area to be deleted, a system error is reported and the pointers are moved to the area most likely to be desired.

The most likely area is guessed from the space requirement. If space is to be reserved, the pointers are positioned to the

replacement point. If no space is requested, the pointers are positioned to the entry following the deleted area or the end of the reduced list.

Parameters: FROM - First ISS entry in deletion area

TO - Last ISS entry in deletion area

SPACE - Table entries to reserve for replacement entries.

SUBROUTINE DELTST (FROM, TO)

Mnemonic Origin: Delete entries from Temporary Symbol Table.

Classification: FFE processing utility

Purpose: Remove entries from the Temporary Symbol Table.

Operation: The provided parameters identify an area of the Temporary Symbol Table to be removed. The indicated entries are deleted by adjusting other entries into the vacant space and adjusting pointers.

Only the main Temporary Symbol Table entries are affected by the deletion. If an entry has an overflow component, the pointers to the overflow element are deleted, but the overflow table is not affected. Overflow table space is not recovered.

Parameters: FROM - First Temporary Symbol Table entry to delete.

TO - Last Temporary Symbol Table entry to delete.

SUBROUTINE DETFA (NAM1, NAM2, CLASS)

Mnemonic Origin: Determine function or array.

Classification: FFE parsing support routine.

Purpose: Distinguish function names from array references.

Operation: A symbolic name is provided which is either a function or an array. The name may be either an external function or statement function; different return codes are required for these two cases.

The determination is made by searching the Symbol Table. If the name is not presently recorded in the Symbol Table, the name is an external function reference. If the name appears in the Symbol Table, the the class is examined. If the name is not recorded as an array, the reference is a function; otherwise, the name is an array.

Note that the name may be recorded as a scalar variable (e.g., an entry in a type statement) on the first reference to the function. In addition, the same character string could be present as a COMMON block label or subroutine name.

Parameters:

NAM1, NAM2	-	Symbolic name of function or array in 2A4 format
CLASS	-	Class code to return indicating either an array, function or statement function.

LOGICAL FUNCTION DIGIT (CHAR)

Mnemonic Origin: Determine if character is a decimal digit.

Classification: System utility.

Purpose: Determine if the symbol presented is a decimal digit.

Operation: The presented symbol is examined to determine if it is a decimal digit.

Since the symbol is in A1 format, the routine is machine dependent upon the character set of the host machine. Care must be taken to avoid comparison between positive and negative valued integers in A format. If the comparison is implemented as a subtraction operation, overflow could result.

Parameters: CHAR - character in A1 format to consider as a decimal digit.

DIGIT - investigation results are returned through the function name.

SUBROUTINE DISFLG

Mnemonic Origin: Display Flag

Classification: Report Generator report processor

Purpose: Produce reports constructed solely from Flag data.

Operation: Upon entry, a message has been detected which is generated solely from data obtained in messages.

The printer is advanced to start the report and the report header line printed. The flag number of the message is examined to determine if a valid Display Report has been requested. The report is then produced.

Note that Display Reports may require more data than can be accommodated in a single message unit. Therefore, the Display Report must retrieve new messages until all data has been processed. Data is complete when a message is received which differs from the current report description.

The only report currently using display format is the cyclic calling hazard report. Routine operation is keyed to this single type. Expansion will require recoding this routine.

Upon exit, the Display Report has been processed and the message buffer is occupied by the first message of the next report.

Special Notes. If the Display Report requires multiple message buffers, redundant message suppression will not operate. Display format analysis routines should not issue redundant information; each copy will produce a new report in this event.

SUBROUTINE DMPAIR

Mnemonic Origin: Dump AIR

Classification: AIR General Purpose Utility

Purpose: Used as aid in debugging of AIP subsystem.

Operation: Prints

1. List Table Map and List Table
2. Control Stack
3. Special Purpose Registers

See also List Table and List Table Map, Control Stack, and /SPEREG/

SUBROUTINE DMPCOM

Mnemonic Origin: Dump COMMON Block Reference Tables

Classification: AIR General Purpose Utility

Purpose: Used as aid in debugging of AIR subsystem.

Operation: Prints COMMON Block Reference Tables (COMMON Block Name Table and Linked List Table).

SUBROUTINE DMPDIR

Mnemonic Origin: Printer dump of Directory contents.

Classification: Maintenance utility.

Purpose: Display the contents of the Directory on a printed display.

Operation: The contents of the Directory and pointers to the Directory entries are displayed in a printed form. The display process interprets the structure of the Directory. Indices of Directory entries are displayed to simplify interpretation of the display. Table and pointer values are not affected by the display process.

SUBROUTINE DMPNOD

Mnemonic Origin: Printer dump of Node Table contents.

Classification: Maintenance utility.

Purpose: Display of the contents of the Node Table.

Operation: The contents of the Node Table are displayed on the printer. The display includes current pointer values as well as contents of the table. Table and pointer values are not affected by the display process.

The structure of the Node Table is interpreted to produce the display. If Nodes have established Successors and/or Predecessors at the time the dump is performed, the Successor and Predecessor Table contents are displayed with entries attached to the appropriate nodes.

SUBROUTINE DMPPTB

Mnemonic Origin: Printer dump of Parsing Tables.

Classification: Maintenance Utility.

Purpose: Display contents of the Parsing Tables (ISS and TSTAB) in a printer display.

Operation: The contents of the Intermediate Symbol String (ISS) and Temporary Symbol Table (TSTAB) are displayed on the print file. In addition to the printed contents of the table, the current pointer values are printed. Indices of entries are printed alongside the entry contents. Table and pointer values are not affected by the display process.

The display is constructed for ease of interpretation. Since not all ISS entries have TSTAB entries, each print line may not have TSTAB contents displayed. Where ISS entries have TSTAB entries associated, the TSTAB contents are displayed on the same line.

TSTAB contents may be directly in the main table entry or located in the TSTAB overflow table. Main entries are printed directly in A format; overflow entries are printed with pointer/counter values extracted followed by the overflow character string.

As a visual aid, the symbol ** is placed alongside the current pointer entries of both TSTAB and ISS value entries.

SUBROUTINE DMPSHT

Mnemonic Origin: Dump System Hierarchy Tables

Classification: AIR General Purpose Utility

Purpose: Used as aid in debugging of AIR subsystem.

Operations: Prints

1. System Hierarchy Tables
2. Inverse System Hierarchy Tables.

SUBROUTINE DMPSYM

Mnemonic Origin: Printer dump of Symbol Table.

Classification: Maintenance Utility.

Purpose: Display the contents of the Symbol Table on the Printer.

Operation: Since the Symbol Table is a hash coded structure, not all entries contain data. Only entries containing nonempty data are displayed. Additionally, the pointers to the Symbol Table are displayed with the table contents.

Where Symbol Table entries are oversized (i.e., require overflow entries), the pointers to the overflow table are printed and the contents of the addressed overflow entries are displayed.

Symbol Table format is interpreted to extract the contents. Table and pointer contents are not affected by the display process.

SUBROUTINE DMPTRP

Mnemonic Origin: Printer dump of Transition Pairs Table (TRIP).

Classification: Maintenance Utility.

Purpose: The contents of the TRIP table are extracted and displayed along with the values of pointers to the table. Extraction requires the interpretation of table construction. Table and pointer values are not affected by the display process.

Flags on the Predecessor entry are separated from the Predecessor specification prior to printing.

SUBROUTINE DMPUSE

Mnemonic Origin: Printer dump of Use Table contents.

Classification: Maintenance routine.

Purpose: Display the contents of the Use Table on the printer.

Operation: The contents of the Use Table are displayed along with the pointer values at the time the print occurred. Pointer and table contents are not affected by the display procedure.

The Use Table structure is interpreted for the display. Since the Use Table entries contain link lists back to Symbol Table entries, the links are traced to provide a more readable output format. In addition, where newstatements begin, a line is skipped to group the uses of one statement in the display.

SUBROUTINE DOTERM (ARRAY, ARRSIZ)

Mneumonic Origin: DO Loop Index used after Loop Terminated Normally

Classification: AIR Query

Purpose: Searches for DO loop index variable used after loop has terminated normally.

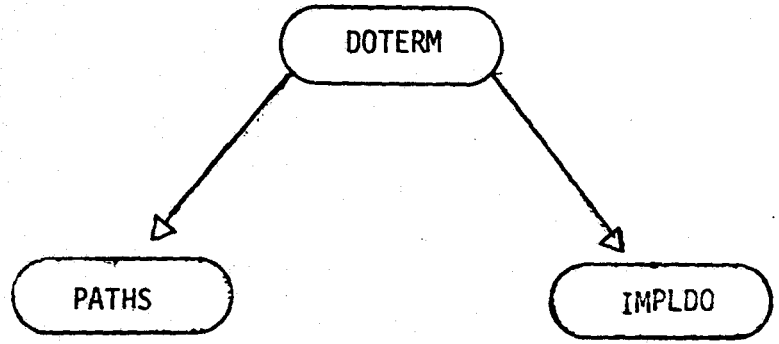
Operations: Array ARRAY has ARRSIZ entries which indicate whether warning flags may be produced for primary or secondary listings.

All paths from end of DO loop to use of DO loop index variable are examined. Program boundaries are not crossed. It is assumed all parameters receive values on other side of program boundary.

Algorithm: See Source Code Listing.

Codes: See Source Code Listing.

See also /PAT/ and PATHS.



Do loop index variable used after loop terminated normally,

excluding General Purpose Utilities.

SUBROUTINE ENDOFD (FILE, POINT, EOFVAR)

Mnemonic Origin: End of Data.

Classification: Control Driver file positioning routine.

Purpose: Move a sequential file to the end of current data.

Operation: A sequential file description is provided by parameters. The file contains (possibly empty) data followed by an end of file mark. This routine is to position the file to the end of file mark so that subsequent data can be appended to the file.

For operational purposes, ENDOFD assumes the file pointer may not be accurate since not all sequential files use the pointer. Additional protection is the initial backspace for the possible position of the file already on the EOF mark. If the pointer is accurate, ENDOFD will update the file pointer in an accurate fashion; otherwise, accuracy of the pointer will not be significant to the calling routine.

After the file is backspaced, sequential reads are performed until the end of file is detected. With each read, the pointer is advanced. When the end of file is detected, the file is backspaced to position the file on top of the file mark, and the pointer is reduced by one to compensate for the backspace operation.

Finally, the end of file variable is reset to indicate the EOF has not yet been reached. (Note: one additional read will cause the EOF to occur.)

Parameters: FILE - I/O unit number of sequential file.

POINT - File pointer associated with the sequential file.

EOFVAR - Variable associated with the sequential file used to record the EOF event.

Designer's Note: Care must be taken that sequential files do in fact contain an EOF mark. FACES is careful to mark all sequential files created, however, errors in the users request may result in unmanipulated files being passed to later processes. On some host machines, this will result in the job being aborted from a read attempt that exhausts the file space.

SUBROUTINE EQUIVL(LISTNO , OVERFL)

Mnemonic Origin: EQUIVALENCE List Construction

Classification: AIR General Purpose Utility

Purpose: Places all names EQUIVALENCED to specified variable into list.

Operation: Algorithm: See Source Code Listing.

Parameters: LISTNO - INPUT

OVERFL - INPUT

Set LISTNO to first available list in List Table Map. If variable indicated by top of Control Stack is in EQUIVALENCE list, place all members of EQUIVALENCE list into list in List Table. If insufficient room in List Table Map or list Table, set overflow flag OVERFL to indicate overflow.

Limitations: If variable in question is array and array elements occur in multiple EQUIVALENCE lists, only one EQUIVALENCE list is assumed to reference array.

SUBROUTINE ERHALT

Mnemonic Origin: Error Halt

Classification: AIR General Purpose Utility

Purpose: Closes files which can be saved when AIR subsystem commits unrecoverable error.

Operations: Close Flag File. Print message. Halt.

SUBROUTINE FAEXP (USECOD, PARENS, INTYP)

Mnemonic Origin: FORTRAN Arithmetic Expression.

Classification: FFE Parsing support routine.

Purpose: Process arithmetic expressions in FORTRAN statements.

Operation: Upon entry, the Parsing Tables are positioned to the first element of an arithmetic expression. The expression is processed recording operand elements with Uses indicated by parameter. As a result of the processing, new symbolic operands are inserted in the Symbol Table; Use references to existing symbols are extended.

The primary complication to arithmetic expression processing is the recursive requirement presented by possible expressions as subscript references or function actual parameters. Additionally, functions are a problem since they represent calls being made to other routines or references to statement function definitions within the same routine. For the entry order in the Use Table to be consistent with the reference order of compiled code, the computation of subscript/parameter expressions and calls to functions should preceed the reference to arrays and Use of the function result.

The arithmetic expression is processed as a simple arithmetic expression up to the first subscripted array reference or function call. The subscript list or parameter list is then processed. If the operand is a function, the table entries corresponding to a function reference are generated before return from the reduction process.

Upon return from the reduction process, the subscript list of arrays are simple operands. The array reference is processed individually

rather than by the simple expression process. Use of the simple sub-expression processing support routine would not permit return on the next array reference; the routine would simply continue processing other expression elements possibly producing faulty results.

After treating the array or function, control continues to process a simple arithmetic expression. Processing continues until a symbol other than an arithmetic operation, operand, or organizing parenthesis is found.

Typing. The typing of the expression is accumulated as processing progresses. The initial type is provided by the calling routine. this parameter is updated as the expression is processed.

Parenthesis Counts. To distinguish organizing parenthesis within the expression from an unbalanced right parenthesis, a count is maintained. For sharing the AEXP routine, the count is physically maintained in the calling routine. This permits interruption of arithmetic expression processing and resumption at a later time.

Cross Reference: See description of arithmetic expression processing in FFE operation description.

Parameters: USECOD - Use code to assign to each member of the arithmetic expression.

PARENS - Parenthesis count for the arithmetic expression

INTYP - Initial type code for arithmetic expression updated as expression is processed.

SUBROUTINE FASS

Mnemonic Origin: FORTRAN Assign Statement.

Classification: FFE Statement Processor.

Purpose: Process ASSIGN statements.

Operation: Most of the effort in processing the ASSIGN statement is the separation of character strings. The blind scan process runs all characters of the ASSIGN statement into a continuous string. These characters must be separated into components of label, character string TO, and variable name.

Character separation is accomplished by copying the text to a local one dimensional array, then extracting characters to detect the end of the statement label and characters of the variable name.

SUBROUTINE FASGMT

Mnemonic Origin: FORTRAN Assignment Statement.

Classification: FFE Statement processor.

Purpose: Control the processing of Assignment statements.

Operation: Upon entry, the Parsing tables are positioned to the first element (assignment variable) of an Assignment statement. The statement may be a simple assignment or the conditional statement of an IF statement.

For the Use table entries to appear in a sequence compatable with the compiled operation, the expression on the right of the equal sign is processed first. To this end, the Parsing Tables are first positioned to the element following the equal sign. The expression on the right of the equal sign is processed assigning Use codes of "input to an assignment statement".

Upon return from the expression process, the Parsing Tables are positioned to the end of the expression or to an error position found in the expression process.

The Parsing Tables are moved back to the assignment variable,

At the end of processing, the Parsing Tables are moved once again to the end of the expression and the process terminates.

Special Conditions. In processing the assignment variable, the Parsing Tables may require reduction to process subscript expressions of array references. This reduction will affect the ISS position recorded for the end of the expression. To sense this movement, the shift in zero level equal sign is used as an adjustment. The equal sign shift will be of the same amount as the end of the statement.

Note that if the statement is aborted by a badly formed expression, processing still returns to process the assigned variable specification. After processing the assignment, the Parsing Tables are restored to the aborted position and processing terminates.

SUBROUTINE FBRLST

Mnemonic Origin: FORTRAN branch list.

Classification: FFE parsing support routine.

Purpose: Process a series of branch specifications separated by commas and terminated by a noncomma.

Operation: Upon entry, the Parsing Tables are positioned to a series of branch specifications (possibly of length one). Each specification is separated by a comma. The list is terminated by a noncomma.

Each branch specification is either a transfer label or an unsubscripted variable which has been set by an assign statement. The branch specifications are extracted and recorded in the Use table. The transitions are recorded in the transition pairs table.

SUBROUTINE FCALL

Mnemonic Origin: FORTRAN CALL statement.

Classification: FFE statement processing routine.

Purpose: Process CALLS to subroutine.

Operation: Upon entry, the Parsing Tables are positioned to a CALL to a subroutine. The subroutine name must be separated from the characters CALL. If a parameter list is present, the list is reduced to simple operands and arrays with simple subscripts by processing out expressions and function calls appearing as actual parameters and array subscripts.

Use table entries are recorded for the subroutine reference and a transition recorded in the transition pairs table for the external reference. The parameter list, if present, is recorded as a list of Uses attached to the subroutine name.

SUBROUTINE FCOMAL (USECOD, FTYP, FCLS, LSTPRO)

Mnemonic Origin: FORTRAN Comma List processor.

Classification: FFE parsing utility.

Purpose: Process FORTRAN comma separated operands which reference program variables and constants.

Operation: FCOMAL is a central processing routine for the FFE servicing many subconstructions in the system. FCOMAL is used for almost all comma-separated forms of lists.

Design Considerations. In FORTRAN, there are many forms of constants and/or variables separated by commas. The members of these lists normally have a uniform meaning. For example, type statements have a series of variables separated by commas following the type specification; all these variables are assigned the same type.

Other comma list forms are uniform up to a break point. For example, in the type declaration statement, the variables may be simple names. Occasionally, an array declaration appears. Similarly, in DATA statements, both the variable list and the data constant lists are comma-separated forms. In the variable list, an array element may be specified by an explicit subscript; in the constant list, a repeat specification may be present.

FCOMAL is designed to process all list element forms until a break is detected. The break is a deviation from the form "operand,". When the break is detected, control is returned to the calling routine for interpretation. For example, when the * form "operand*" is found in a data constant list, control is returned to the calling routine

for interpretation. Similarly, if an (is found in an I/O list, control is returned.

The calling routine controls action at the breakpoint. By parameter, DCOMAL is informed if the last operand of the break is to be processed. For example, in an I/O list, if the form "name(" is found, the name is still an I/O variable, so the calling routine would request the name be processed before returning. If the form "(" is found in the comma list without a preceeding name, permission to process the last operand would have not effect-- the break did not occur on an operand.

On the other hand, when processing the constant list of a DATA statement, the appearance of "operand*" would require different action. The operand is a repeat specification, not a data value assignment. In this case, the calling routine would request the operand not be processed at breakpoints. The operand is returned for treatment in the calling routine.

Operational Sequence. To accommodate superficial differences in constructions permitted, FCOMAL will permit a leading comma to begin the list. This simplifies the housekeeping activities in the parse. If a leading comma appears, it is discarded.

From that point, operands are processed so long as the pattern "operand," is found. When either the operand or comma fails to appear, the break has occurred. If the break occurs on an operand entry, the calling routine control option dictates whether the operand is to be processed.

All processed operands are recorded with use, type, and class under the calling routine's control. The calling routine may default the type and class to values implied by the ISS code or force a

particular type and/or class on list members.

At the end of processing, the Parsing Tables are positioned to the last unprocessed ISS entry.

Parameters: USECOD - use code to assign to list members.

FTYP - type code specification provided for assignment to list members.

FCLS - class specification provided for assignment to list members.

LSTPRO - indicator as to whether the last operand of the list is to be processed at the breakpoint.

SUBROUTINE FCOMON

Mnemonic Origin: FORTRAN COMMON statement.

Classification: FFE statement processor.

Purpose: Process FORTRAN COMMON statements.

Operation: The processing of COMMON statements is complicated by the blind scan run-on of character strings. If implied blank COMMON is referenced, the character string of the first variable name will be run on with the characters COMMON.

To determine if this has occurred, the character position following COMMON is examined; if it contains a blank, there is no "hidden variable".

If blank COMMON is referenced, the first variable name is extracted and replaces the Parsing Table entry to normalize the variable representation as a series of variable names separated by commas. The list of variables assigned to blank COMMON is then processed until the end of statement or next COMMON label is encountered.

Of implied blank COMMON is not referenced, then control passes immediately to the series of explicit label COMMON specifications. COMMON labels (including the form // indicating explicit blank COMMON) are processed. The label is recorded and the variable list. This process is repeated until no more COMMON block specifications are found in the statement.

Designer's Comments: A methodology was initially sought for combining the processing of implied blank COMMON with explicit labeled

COMMON. However, algorithms which accomplished this objective were judged more difficult to comprehend. For this reason, the special case processing of blank COMMON was chosen for implementation.

SUBROUTINE FDATA

Mnemonic Origin: FORTRAN DATA statements.

Classification: FFE statement processor.

Purpose: Process DATA statements.

Operation: The first step in processing DATA statements is to determine if the first variable name has been run-on with the characters DATA. If the first DATA variable list begins with a variable name, this condition will exist. If the first variable list begins with an implied DO loop, run-on will not occur.

If the variable name is run-on with the characters DATA, the name is extracted and Parsing Table entries replaced to normalize the list format.

The DATA specification is then processed as a series of variables in the form of an I/O list, followed by data assigned constants. The constants are bounded by a pair of /'s. The comma separating the last / and the next variable of a DATA variable list is optional.

Processing terminates upon finding a symbol after the last / other than a variable or comma.

SUBROUTINE FDATL

Mnemonic Origin: FORTRAN Data List.

Classification: FFE parsing utility.

Purpose: Process constant specification list of initial values.

Operation: Upon entry, the parsing tables are positioned to the first / delimiting a list of constant specifications. The list may contain simple constants, signed constants, or repeat specified constants. Constant specifications are separated by commas.

The comma list processor is used on the operand list. The processor is instructed to force all list components to a constant class. Further, the processing of the last entry at breakpoints is inhibited to permit the contingency of repeat specifications. This mode of operation leaves FDATL with the responsibility of directly processing the last constant itself.

Return from the comma list process may be caused by an explicit sign, the appearance of "operand*", or the last list element. If a sign appears, the sign is discarded and the signed operand processed. If a repeat specification is detected, the repeat count and operand are processed. Termination occurs when no comma is found after a list entry. The trailing / is not used for termination since a keypunch error omitting the / would cause a processing failure.

Special Remarks: In some FORTRAN dialects, variable like constants can be used in Data Lists (e.g., T or F for logical TRUE or FALSE). FDATL will force the class of these entries to constant permitting their recognition by other routines.

SUBROUTINE FDIME

Mnemonic Origin: FORTRAN DIMENSION statement.

Classification: FFE statement processor.

Purpose: Process DIMENSION statements.

Operation: Processing DIMENSION statements requires separating the first variable name from the characters DIMENSION. The extracted variable name is aligned on the first character and replaced in the Parsing Tables. From this point, the statement is processed as a variable declaration list. Uses are not recorded for appearance in a DIMENSION statement; only the Use as an array declaration is required.

SUBROUTINE FDO

Mnemonic Origin: FORTRAN DO statement.

Classification: FFE statement processor.

Purpose: Process DO statements.

Operation: In processing DO statements, the characters DO, statement label specification, and index variable must be separated from the single character string developed by the blind scan process. Use of the statement label and graphical transition for a DO statement are recorded.

The index variable is extracted and recorded, followed by processing of the DO loop control parameters.

SUBROUTINE FDORNG

Mnemonic Origin: FORTRAN range variables.

Classification: FFE parsing support routine.

Purpose: Process the range control variables of DO and implied DO constructions.

Operation: Control enters with the Parsing Tables positioned to the first variable (initial value specification) of a DO control construction. The control specifications may be signed or unsigned constants or variables separated by commas. The increment specification is optional.

To process the optional length of DO control specifications, the processing is constructed in the form of a loop. The loop is terminated when a comma is not found after a control specification or the increment specification has been processed.

If a sign is found leading the control specification, the sign is discarded. The Use code assignment is dependent upon the numerical adjacency of Uses for initial, terminal, and increment specifications. The Use code for the next control specification is developed by incrementing the last specification by one.

Control returns to the calling routine with the Parsing Tables positioned to the entry after the control specification.

SUBROUTINE FELxxx(LC, ROW, A)

Mnemonic Origin: Fetch Element from Table xxx

Classification: AIR General Purpose Utility

Purpose: Allows access to values of elements in table xxx.

Operation: xxx may be one of the following:

COM, DIR, IS, ISD, LIN, LIS, MAP, NOD, PRE, SH, SHD,
STK, SUC, SYM, USE.

Parameters: LC - Input
ROW - Input
A - Output

Element in row ROW and in logical column LC of Table xxx and associated information is placed in array A. Element's type (alphanumeric or integer) is placed in element one of array A. Element's width in computer words is placed in second element of array. Element is placed in remaining elements of array, left-adjusted.

see also "AIR Abbreviations" and /SPEREG/

SUBROUTINE FENTRY

Mnemonic Origin: FORTRAN ENTRY statement.

Classification: FFE statement processor.

Purpose: Process ENTRY statements.

Operation: Control enters with the Parsing Tables positioned to a secondary entry point specification for either a subroutine or function. The entry point name is extracted from the character string run-on by the blind scan process.

The current module specification is examined to determine if a subroutine or function entry point is specified. This information will influence the type code assigned to the entry point name and class code of secondary entry parameters (if any).

The entry point name is recorded in the Symbol Table and a secondary entry point recorded in the transition pairs table. (Note: If the entry point is a function entry point, the type code of the name is determined independently from the primary entry point name.)

The parameter list, if specified, is then processed.

SUBROUTINE FEQUIV

Mnemonic Origin: FORTRAN EQUIVALENCE statement.

Classification: FFE statement processor.

Purpose: Process EQUIVALENCE statements.

Operation: Control enters with the Parsing Tables positioned to the start of an EQUIVALENCE statement. The statement is composed of the character string EQUIVALENCE followed by a series of equivalence group specifications. EQUIVALENCE groups are sets of variable specifications enclosed in parentheses. The groups are separated by an optional comma.

Within the group, individual specifications may be subscripted or unsubscripted variable references. The processing of a group is terminated by the absence of a comma following an element of the group. The test for a right parenthesis is not used to protect against a keypunch error in which the parenthesis might be omitted.

Additional error tolerance is permitting the absence of a right parenthesis if the next symbol is a left parenthesis. To guard against the error of only the word EQUIVALENCE without any specified group, a flag is maintained to indicate whether any EQUIVALENCE groups were processed.

SUBROUTINE FEXTRN

Mnemonic Origin: FORTRAN EXTERNAL statement.

Classification: FFE statement processor.

Purpose: Process EXTERNAL statements.

Operation: Upon entry, the EXTERNAL statement has the first external run-on with the keyword characters. Since the keyword EXTERNAL has 8 characters, alignment is not necessary; only skipping the first 8 characters is required. The list of names is then processed as a simple comma separated list of entries.

SUBROUTINE FFE

Mnemonic Origin: FORTRAN Front End.

Classification: Primary control for FORTRAN processing.

Purpose: Process a series of FORTRAN modules from the Source Code Input File, producing Analysis Tables and Source Code Catalogue entries.

Operation: Upon entry, the processing of a series of FORTRAN modules from the Source Code Input file is required. Linkage must be established to incorporate new modules on the Analysis and Source Code Catalogue files. This linkage of library information is accomplished via the Global Header.

The Source Code Catalogue index for the next module is acquired from the Global Header. The module relative card counts are initialized. The last module number used is acquired from the Global Header (this establishes the next Table file record available).

Modules are then processed in a cyclic fashion until the end of the Source Code Input file is detected. Note that processing accommodates the contingency of a module not being produced. This might occur if the Source Code Input file was empty or extraneous cards are found at the end of the file.

After a valid module has been processed, the Directory contents are updated and maintenance printing processed if active. Results are then stored on the Analysis Table file.

After all modules have been processed, maintenance printing of the Directory contents is performed if active, and the file information for Source Code Catalogue and Analysis Table files is recorded in the Gobal Header.

SUBROUTINE FGOTO

Mnemonic Origin: FORTRAN GO TO statements.

Classification: FFE statement processor.

Purpose: Process forms of GO TO statements.

Operation: Upon entry, a GO TO statement form has been identified. The form may be a simple Unconditional GO TO, Assigned GO TO, or Computed GO TO.

To process the statement, the branch target must be extracted from the character string GOTO. If no branch target is found following the GO TO, the form is a Computed GOTO; the left parenthesis caused the building of character string entries to terminate with the characters GOTO.

In the Computed GO TO form, the potential transfer target list is processed, followed by the GO TO index variable. The comma separating the branch list from the index variable is optional.

If a nonblank name is found after the GOTO character string, the form is either an unconditional branch to a statement label or an Assigned GO TO. This distinction is made based on the characteristics of the branch target specified. If the branch target is a numeric entity, the Unconditional form is recognized; if the branch target is a variable name, the Assigned form is present.

In Assigned GO TO forms, the list of potential branch targets is optional. If the branch target list is specified, the list of statement labels will be attached as successors to the GOTO statement. The comma separating the branch switch variable from the list of successors, is optional.

SUBROUTINE FIDNO (CNT, KU)

Mnemonic Origin: FORTRAN Identification Number.

Classification: FFE parsing support function.

Purpose: Process optional identification indicators attached to FORTRAN statements like PAUSE, STOP, etc.

Operation: Upon entry, a statement form has been recognized which might have an identification attached. The statement is such that the character string returned from the scan process has run on the identification specification. FIDNO extracts the identification from the character string and records a Use specified by the calling routine based on properties of the identification.

The parameters indicate how many leading characters precede the identification. These characters are bypassed and the identification position is examined. If blanks are found, no identification is recorded; the identification has been optionally omitted.

If nonblank characters are found, the characters identifying the statement (up to 8 characters) are extracted. If an alphabetic identification is found, the symbol is recorded as a scalar variable used as specified by the parameter. If a numeric symbol is found, the symbol is recorded as an integer constant used as specified by parameter.

Parameters: CNT - integer count of the number of leading characters which precede any identification specification.

KU - Use code specification to assign to the identification if one is found.

SUBROUTINE FILOPN

Mnemonic Origin: File Open.

Classification: Control Driver File manipulation routine.

Purpose: Perform opening activities for files to isolate initial file positioning from required run configuration.

Operation: FILOPN is a host machine dependent routine for positioning files and opening activities required on a particular host. Sequential files produced by FACES are rewound to permit internal sensing of file status. The end of file indicators for sequential files is cleared. The Source Code Input File is not positioned to avoid circumventing user control of input source code to be analyzed.

Random file routines are initialized in this routine if necessary.

SUBROUTINE FILCLS

Mnemonic Origin: File Close.

Classification: Control Driver File manipulation routine.

Purpose: Perform closing actions necessary for termination of run.

Operation: FILCLS is a machine dependent routine provided to centralize any terminal file activity required on a particular host machine. Actions performed by FILCLS are intended to secure data generated by the FACES run and prepare data files for saving action or manipulation external to FACES.

SUBROUTINE FIF

Mnemonic Origin: FORTRAN IF statement.

Classification: FFE Statement Processor.

Purpose: Process IF statements.

Operation: Upon entry, a FORTRAN IF statement has been identified. The IF statement is of the form,

Arithmetic IF with three branch transfer,

Logical IF with two branch transfer,

Logical IF with conditional statement specified .

Note that the IF statement may be identified by the Keyword processing section or from the Zero Level Equal Sign processor. If the form is a conditional Assignment statement, the entry will be via the Zero Level Equal Sign processor; otherwise, entry will be from the Keyword processor.

The conditional expression is processed until a balancing right parenthesis is found or an unknown symbol is encountered. The conditional expression is viewed as a series of arithmetic expressions separated by possible relational operators, organized by parentheses. Some complication is presented by parentheses in the conditional expression. Since FACES processes relational expressions only in IF statements, relational operators are processed as they appear in the left to right parse. Since no global count of parentheses is maintained, care must be taken to account for all possible configuration of parentheses. Observe that parentheses may be used to:

1. Organize the computation of an arithmetic expression.
2. Organize the relation application between two arithmetic expressions.
3. Organize the computation of logical expressions from relational results or logical variables.
4. Indefinite levels of superfluous parentheses could be present.

To illustrate operation, consider processing the form

IF((((A+1 .GT. B).OR.C.LT.5).AND.(LOG)))

Note that a superfluous pair of parentheses is used in the conditional expression. As will be obvious, the scanning of expressions will result in an execution profile quite different from traditional expression parsing. The example statement is processed as follows:

1. Processing the expression begins on the second left parenthesis. An arithmetic expression is processed up to the relation .GT. Control returns with a parenthesis count equal to 3. (The "(" of "IF(" is not included in the count).
2. The relation .GT. is processed.
3. Control returns to arithmetic processing which proceeds to the relational operator .LT. The elements B, C, and operator .OR. are processed. The parentheses count upon return is equal to 2.
4. The relation .LT. is processed.
5. Control returns to the arithmetic process and the elements 5 and logical variable LOG are processed. Control returns with the Parsing Tables positioned to the last right parenthesis with a parentheses count of 0.

After processing the conditional expression of the IF, the conditional action is examined. If the form "operand," is found, a branch specification is assumed and the branch list processed. (Note: the number of branch targets specified are not checked in the process.)

If a branch list is not found, a conditional execution is assumed and control is returned to the calling routine with the Parsing Tables positioned to the first entry of the conditional statement for later processing.

SUBROUTINE FIMP

Mnemonic Origin: FORTRAN IMPLICIT statement.

Classification: FFE statement processor.

Purpose: Process IMPLICIT statement and adjust typing of previously recorded symbols to conform to IMPLICIT requirements.

Operation: The IMPLICIT statement is composed of the character string IMPLICIT followed by a comma separated series of type specifying definitions for leading letters of variable names. Each type specification is of the form,

type indicator (letter specification)

where type indicator is a character string indicating the type to be associated with the letters specified, and letter specification is a comma separated list of either single letters or an inclusive range of letters (e.g., "A-D").

In processing the statement, the type indicator is extracted and identified. The type indicator establishes both the type code for variables and the length of the type indicator expressed as the number of packed words (e.g., INTEGER is a type indicator requiring 1 word of storage).

The type identification is bypassed and the list of letter specification processed. Each letter specification causes the typing vector to be modified from the normal default type associated in FORTRAN. Type vector positions are modified by the type specification until a comma is not found between the end of the letter specification and the next type identification. The search for comma absence is used

to control error processes in the event of a keypunch error.

After establishing new types for alphabetic leading characters, the Symbol Table contents are reviewed. Any typable entry which does not appear in an explicit type statement is retyped.

SUBROUTINE FIOARG

Mnemonic Origin: FORTRAN I/O control argument list.

Classification: FFE parsing support routine.

Purpose: Process the I/O control list of I/O statements.

Operation: ANSI Standard I/O requires specification of I/O unit and format; many FORTRAN dialects also include End of File detection and Error Recovery processing in this control list. Of the potentially specified controls, only the I/O unit number is mandatory.

The I/O unit may be specified as either a constant or variable. To accommodate FORTRAN dialects with direct access I/O, a record within the unit may be specified. The unit record specification is restricted to a simple variable or constant.

The format specification is optional. If a format is specified, it may be either the statement label of a FORMAT statement or an array. Since the array may not appear with a subscript, FACES will accept a scalar variable as a FORMAT specification.

Specification of End of File and Error condition processing take the form of "END= branch target" and "ERR= branch target". The branch target may be either a statement label or assigned variable. To process the exception list, the control specification is first reviewed for ERR and END specifications. If they are detected, the Parsing Table positions of the branch target specification are recorded. At the end of the review, the actions for END and/or ERR are processed. This approach was selected so the branch order would always be END/ERR.

Since exception lists may be machine dependent, FIOARG is designed to report exceptions not processed and attempt to skip unknown forms.

Statement abortion is not performed since most of the important information is in the I/O variable list which follows the control specification.

Designer's Note: A great deal of effort is expended to normalize the order of exception processing. For the most part, this is wasted energy. Since the exceptions are optional, the order of exceptions is still unpredictable; from the tables generated, a single exception specification cannot be identified as an END or ERR branch. This area is candidate for change in future versions.

SUBROUTINE FIOVAR

Mnemonic Origin: FORTRAN I/O Variable List.

Classification: FFE parsing support routine.

Purpose: Process I/O form variable lists in I/O statements and DATA statements.

Operation: Upon entry, an I/O form list of variables is to be processed. The list may contain subscripted and unsubscripted variables, implied DO loops, and superfluous grouping parentheses. Note that the appearance of a left parenthesis does not guarantee an implied DO loop; The ANSI Standard permits parentheses grouping without an implied DO.

Elements of the I/O list are assumed to be separated by commas. The absence of a comma after a list element signals the end of the list. The list is processed as a comma list with the processing of entries inhibited at breakpoints. This restrictive action is required to prevent the index variable of an implied DO from being recorded as an I/O variable in the list. As a result, FIOVAR is responsible for processing the last I/O variable itself.

If control returns from the comma list processor positioned to a variable entry, one of the following situations is possible:

1. The variable is the last variable of the I/O list.
2. The variable is subscripted.
3. The variable is the index variable of an implied DO loop.

If the return is caused by a right parenthesis, this is not the right parenthesis of an implied DO; rather, the parenthesis is a superfluous parenthesis used in organizing the list. The right parenthesis of the implied DO is processed in the DO specification case.

Special Notes:

The processing of parentheses is rather strange in FIOVAR. The opening left parenthesis is processed by code located at the top of the processing loop. The corresponding closing right parenthesis is processed either as the right parenthesis of an implied DO or as an independent right parenthesis organizing the I/O list for the user.

Notice that FIOVAR deviates from other processing routines in that the I/O list is assumed to be present. Calling FIOVAR with constructions having empty I/O lists (i.e., no list specified) will cause processing errors. This feature should be amended in future versions.

Table entries produced by FIOVAR have a major failing for analysis of implied DO cases. The use of the DO loop index variable as an input appears before the assignment of value in the Use Table. This should be repaired in future versions.

FUNCTION FLD(POS, NOBITS, WORD)

Mnemonic Origin: Univac's FLD Function

Classification: AIR General Purpose Utility

Purpose: Pulls out bits from single word.

Operations: NOBITS bits are pulled from word WORD, starting at bit position POS, and are returned to calling routine, right-adjusted.

Left-most bit in word is defined as having position zero.

Parameters: POS - Input

NOBITS - Input

WORD - Input

FLD - Output

INTEGER FUNCTION FLGFUL (FLAG, FLGSIZ, FWORD)

Mnemonic Origin: Flag a Full word integer.

Classification: FFE bit manipulating utility.

Purpose: Construct a full word integer composed of positive data in LSB positions and flag bits in MSB positions.

Operation: A bit pattern is to be inserted into the MSB bit positions of a full integer word to construct a result.

Parameters describe the flag size of the LSB's of the flag pattern to combine with the LSB's of integer data. This operation requires machine dependent bit manipulations.

The MSB's of the integer data are cleared to zero value to accommodate flag information. Then the LSB's of the flag information are shifted to the MSB position and combined with the full word data to produce the result.

Parameters: FLAG - full word integer containing flag bit pattern in the LSB positions.

FLGSIZ - flag size measured in number of bits.

FWORD - full word integer data.

FLGFUL - the constructed full word is returned through the function name.

INTEGER FUNCTION FLGHLF (FLAG, FLGSIZ, HWORD)

Mnemonic Origin: Flag a Half Word integer.

Classification: FFE bit manipulating utility.

Purpose: Construct a flagged half word from a half word of data and a flag specification.

Operation: FLGHLF constructs a full word integer which contains a flagged half word of data in the LSB bit positions. The MSB positions of the integer contain zeroes.

The data is constructed by clearing the upper half word of the provided data along with MSB of the lower half word which will accommodate the flag. The size of the flag is indicated by parameter. The flag to insert is located in the LSB positions of the Flag specification.

The actual flag bits are positioned to the MSB bit positions of the lower half word and inserted in the positions prepared.

Parameters:

FLAG - contains flag data to be inserted in the MSB of the result half word. Data bits are found in the LSB positions.

FLGSIZ - Size of the flag field expressed in number of bit positions.

HWORD - full word integer containing data in the lower half word to be used in construction of the result.

FLGHLF - results returned through the function name.

LOGICAL FUNCTION FNDDIR (NAME)

Mnemonic Origin: Find Directory entry.

Classification: FFE Table search routine.

Purpose: Search for the specified name in the Directory and position the Directory pointer to the matching or insertion point.

Operation: A module name (possibly assigned by FACES) is presented. If the name is found in the Directory, this event should be reported and the Directory pointer positioned to the matching entry. If the name is not found in the Directory, the absence should be reported and the Directory pointer positioned to the proper insertion point.

FNDDIR is constructed to give an alphabetic order to the Directory entries. The Directory may be empty on the first insertion of a name.

If the Directory is empty, the no match condition is recognized and the insertion point is the first Directory position.

If the Directory is not empty, the contents of the Directory is searched for the module name. If a matching entry is found, the match condition is recognized and the Directory pointer is positioned to the matching name.

If the search process discovers an entry aplphabetically ahead of the name used in the search, a no match condition is recognized and the insertion point of the alphabetic entry is established.

Parameters: NAME - Module name to search for in the Directory in 2A4 format.

FNDDIR - the match/nomatch result is returned through the function name.

Special Notes: Alphabetic order is host machine character code dependent. Currently, all Directory entries begin with alphabetic characters. One problem was discovered during maintenance where a nonalphabetic character was used as an assigned name. The nonalphabetic character caused a positive and negative A format item to be compared in the relational test. This resulted in an overflow condition which was not reported at execution time, yet resulted in an erroneous branch being taken. As a result, the alphabetic order of the Directory was destroyed and insertion processing collapsed from that point.

If the Directory is spanned without finding an alphabetically preceeding entry or a match, the no match condition results and the insertion point becomes the next nonempty Directory entry. Note that the Directory bounds are not checked in establishing the insertion point for this event. Checking of array space should be performed by the insertion routine.

LOGICAL FUNCTION FNDSYM (NAME, LENGTH, TYPE, CLASS)

Mnemonic Origin: Find Symbol Table entry.

Classification: FFE table insertion support routine.

Purpose: Search Symbol Table entries for the specified symbol and set the Symbol Table pointer to the insertion or match position.

Operation: A symbol specification is presented by parameter. If the symbol matches an existing Symbol Table position, the matching result is reported and the Symbol Table pointer positioned to the matching entry. If the symbol is not present in the Symbol Table, the no match condition is reported and the Symbol Table pointer positioned to the proper insertion point for the symbol.

In addition to the character string for the symbol, a type and class specification are provided in examining Symbol Table entries. The value of the specifications may be either an established code or the value 0. The value 0 means the corresponding symbol qualifier is to be ignored in the search. A nonzero value means the Symbol Table position must match the qualifier in addition to the character string to qualify as the desired entry.

The hash coded entry point is computed based on the first 8 characters of the symbolic name. The Symbol Table entries are then searched in modulo table length fashion until a matching entry is found, an empty position encountered, or all table entries have been searched.

A matching entry must satisfy the following requirements:

1. The length must be equal to the specified length.
2. The character content must be the same.

3. The class code must be compatable with the specified class.

4. The type code must be compatable with the specified type.

These conditions constitute logical AND operations. The AND is implemented as a series of successive branches. The logical outcome is accumulated as the tests are performed; any failure causes the table position under examination to not qualify.

If the entire table is searched without finding an empty position or matching entry, the Symbol Table is full; no space remains for entry insertion. The no match condition is returned and the Symbol Table pointer is set to 0 to indicate no space is available for insertion.

Since the Symbol Table is positioned by the search process, the match condition simply requires maintaining the current Symbol Table pointer value to indicate the matching position. Similarly, if an empty table position halts the search, the current pointer value is the appropriate insertion point for the symbol.

If the symbol is oversized (i.e., will not fit in the main Symbol Table position), the Symbol Overflow pointer may be set. If a match occurred, the Symbol Overflow pointer is set to the position indicated by the matching position. If no match was found, the overflow pointer is not set.

Parameters: NAME - Symbolic characters of the symbol being sought in A4 format.

LENGTH - Length of the symbolic name expressed in the number of integer words required to hold the character string.

TYPE - Type specification for matching condition. Contains either the type code for an acceptable entry or the value 0 (ignore type in the match process).

CLASS - Class specification for matching condition. Either a class code or the value 0 for ignoring class in the search.

FNDSYM - The match/no match condition is reported through the function name.

INTEGER FUNCTION FNDTST (ISSLOC)

Mnemonic Origin: Find Temporary Symbol Table position.

Classification: FFE Parsing Table positioning routine.

Purpose: Identify the Temporary Symbol Table position corresponding to the indicated ISS table entry.

Operation: FNDTST is used to determine Temporary Symbol Table positions based upon ISS entries. By convention, the two tables are maintained in synchronized fashion; therefore, the required entry sought is normally not the currently addressed table entry.

Temporary symbols are recorded only for character strings corresponding to variable names, constants, and FORTRAN keyword phrases. The Temporary Symbol Table entries lead the ISS entries by one position. (See discussion of Parsing Tables.)

FNDTST operates on the assumption that the two tables, ISS and TSTAB, are properly synchronized. Based upon the current positonal relationship, the Temporary Symbol Table position for the indicated ISS position is produced. This operation is performed by using image pointers to TSTAB and ISS. Modification of table contents and pointer positions are expressly avoided.

The direction and number of ISS positions between the current position and the desired position are determined. The value of the TSTAB image pointer is then adjusted to the proper value by reviewing ISS position contents. For ISS entry in which a TSTAB entry is recorded, tne image pointer value is adjusted. The final value is determined when the image pointer to ISS is equal to the desired position indicated by parameter.

Note that the "appropriate" position for TSTAB may be an entry which leads the ISS position. For example, if the specified ISS position were to contain a "(", there is no directly associated TSTAB entry; rather the TSTAB entry would be the next variable or constant to be found after the left parenthesis.

In addition, the leading convention between tables allows the developed pointer value to exceed the last nonempty position value by one. This would result, for example, when developing the TSTAB position for a special symbol, say a ")", which appeared after the last variable or constant in the statement. That is, all TSTAB entries were associated with preceeding ISS entries.

Parameters: ISSLOC - position of ISS for which the TSTAB pointer value is desired.

FNDTST - pointer position is returned through the function name.

LOGICAL FUNCTION FNDUSE (USECOD)

Mnemonic Origin: Find Use of current symbol.

Classification: FFE table searching routine.

Purpose: Determine if the currently selected symbol has been used in the specified fashion.

Operation: Upon entry, the symbol table is positioned to a symbol under investigation. The decision is required as to whether the symbol is used in a way indicated by parameter.

The last Use of the symbol is acquired and the chain of Uses from the last to the first Use is examined. If the specified Use code is found in this list, the TRUE condition is returned and the Use table is left positioned to the first Use encountered (i.e., the most recent occasion of the Use).

If the Symbol Table is in an overflow state, no active symbol is recorded; therefore, no Use is possible. The FALSE result is returned in this case. Similarly, if no Uses for a recorded symbol have been made, the FALSE result is returned. The Use table pointer is not positioned in these cases.

If a Use list is found but the specified Use is not encountered on the list, a FALSE result is returned and the Use Table pointer is left positioned to the first Use of the symbol.

Parameters: USECOD - use code to look for in the symbol's Use list.
FNDUSE - results of the search returned through the function name.

SUBROUTINE FPARLS (KU, KT, KC)

Mnemonic Origin: FORTRAN Parameter List.

Classification: FFE parsing support routine.

Purpose: Process forms of FORTRAN constructions consisting of a comma list enclosed in a parenthesis pair which require intra-statement Begin/End list brackets.

Operation: Upon entry, the Parsing Tables are positioned to the left parenthesis of a "(comma list)" structure. These constructions are dummy parameter lists for functions, statement functions, and subroutine definitions.

The Symbol Table has previously been positioned to the name of the routine so the list generated becomes attached to the name as a dependent list.

A Begin List Use code is recorded, followed by the list of actual parameters. Finally, an End List is recorded.

Parameters: KU - use code to assign to comma list members.

KT - type specification (possibly zero) for comma list members.

KC - class specification (possibly zero) to assign to comma list members.

Special Notes: The End list is recorded even if the statement is aborted for an unrecognized actual parameter list structure. This may result in a system anomaly since abortion clears the Begin/End list stack.

SUBROUTINE FPRNCL (USE, TYP, CLS)

Mnemonic Origin: FORTRAN Parentheses Comma List.

Classification: FFE parsing support routine.

Purpose: Process comma lists enclosed in parentheses.

Operation: FPRNCL processes FORTRAN constructions consisting of simple operands separated by commas, enclosed in a pair of parentheses. The elements of the list may not contain expressions, function calls, or any form more complex than simple operands.

FPRNCL is used to process the following forms:

1. Simple operands of array subscript references.
2. Simple operand forms of function and subroutine references.
3. Simple operand forms of function and subroutine parameter declarations.
4. Simple operand forms of array dimension declarations.

Upon entry, the Parsing Tables are positioned to the left parenthesis of the list. The list is processed as a simple comma list. Control returns with the Parsing Tables positioned to the element after the enclosing right parenthesis.

Parameters: USE - Use code to assign to list members.

TYP - Type specification for list members.

CLS - Class specification for list members.

SUBROUTINE FPROG

Mnemonic Origin: FORTRAN PROGRAM statement.

Classification: FFE statement processor.

Purpose: Process FORTRAN PROGRAM statements.

Operation: The program name is separated from the character string PROGRAM. The program name is established as the module name. The name is also recorded in the Symbol Table. The declaration Use of the name is recorded in the Use Table. An initial entry into the module is recorded as a transition. If a qualifier list enclosed in parentheses is found, the list is skipped; this information is not used in analysis.

SUBROUTINE FPRPU

Mnemonic Origin: FORTRAN PRINT or PUNCH statements.

Classification: FFE statement processor.

Purpose: Process PRINT and PUNCH statements.

Operation: Both PRINT and PUNCH are processed by this one routine since both are output type I/O statements, and the keywords are both the same length (5 characters).

The FORMAT specification is extracted from the run on character string PRINT or PUNCH. The FORMAT may be either a statement label or a variable name. The distinction is made by examining the first character of the specification; a numeric character indicates a statement label.

The FORMAT specification is recorded, then the I/O list is processed if one exists. The presence of an I/O list is determined by detecting a comma following the FORMAT specification.

SUBROUTINE FREAD

Mnemonic Origin: FORTRAN READ statement.

Classification: FFE statement processor.

Purpose: READ forms processed are limited by the ANSI Standard I/O forms composed of an I/O control list enclosed in parentheses followed by an optional variable list. If the leading left parenthesis is not found, a machine dependent form of READ is present; this form is not processed.

The I/O control arguement list is processed, followed by the optional variable list of I/O variables.

SUBROUTINE FSIMEX (USECOD, INTYP, INPRN, SIMPLE, OUTTYP, PARENS)

Menmonic Origin: FORTRAN Simple arithmetic expression.

Classification: FFE parsing support routine.

Purpose: Process simple forms of arithmetic expressions, returning control if a complex form is found.

Operation: FSIMEX is intended to process arithmetic expressions whose most complex component is an array reference with simple operand (variables or constants) subscripts. FSIMEX is informed by the calling routine if subscripts of arrays have been reduced. If they have, the array reference is processed; otherwise, control is returned on encountering an array.

FSIMEX is coded as a small control loop at the top with actual manipulation service provided by Case processes for components of arithmetic expressions. The expression is scanned rather than parsed (i.e., operator precedence rules are ignored). Control returns to the calling routine when one of the processing cases encounters a symbol which indicates the end of the expression or a form is encountered which is beyond the capacity of FSIMEX. All case processes return to the common return point.

Simple variable or constant operands are processed as components of the arithmetic expression. The type contribution of the operand is accumulated in the expression result type.

If an operand is encountered, the parsing symbol is examined for the form "Variable(". If this form is found, either an array reference or function reference has been detected.

If a function or array is detected, the control parameter to FSIMEX is examined. If the subscript list or function actual parameter list has not yet been reduced, control is returned to the calling routine with the Parsing Tables poisitioned to the function or array name. Functions are treated elsewhere. Therefore, upon return to FSIMEX the only possible construction with form V(is an array reference with reduced subscripts. The reduced array form is processed with the array variable type contributing to the type accumulation of the expression.

Arithmetic operators are simply skipped. Note that the multiplication symbol ** is treated as two separate symbols. This is not significant since the order of operands and precedence of operators is not considered in the scan.

Parentheses processing requires distinguishing organizing parentheses from terminal parentheses. A right parenthesis is terminal if FSIMEX has not processed a corresponding left parenthesis. To accomplish this, a parentheses count is maintained. Only the organizing parenthesis of arithmetic expressions is considered in the count (i.e., parentheses of array references and function calls are not included in the count). The parentheses count is a positive integer count with zero having the meaning that all parentheses have been balanced. If a left parenthesis is detected, the count is advanced; if a right parenthesis is found and the current count is already zero, control is returned to the calling routine with the Parsing Tables positioned to the right parenthesis. This case is the detection of an external closing parenthesis of a more global structure (i.e., the closing parenthesis

of a subroutine call in which an actual parameter is an arithmetic expression).

Control is also returned if an entry is found in the Parsing Tables which is not an arithmetic expression component (e.g., a comma).

To permit interruption of processing and resumption at a later time, the parentheses count and type accumulation are maintained in the calling routine. This implementation also permits FSIMEX to be used for both arithmetic expression processing and to process arithmetic expressions used as subscript or function actual parameters. (See arithmetic expression processing discussion in FFE.)

Parameters:

- USECOD - Use code to assign to operand elements of the arithmetic expression.
- INTYP - Type specification for expression upon entry (may be 0 before any arithmetic expression operands processed).
- INPRN - Parentheses count upon entry.
- SIMPLE - Logical control parameter set by calling routine to indicate whether function parameters and array subscripts have been reduced.
- OUTTYP - Accumulated type code from expression operands processed and initial type indication.
- PARENS - Parentheses count on return to calling routine.

SUBROUTINE FSTFUN

Mnemonic Origin: FORTRAN Statement Function.

Classification: FFE statement processor.

Purpose: Process statement function definitions.

Operation: The statement function name and parameter list are processed. Parameter list elements are recorded with a class code of "statement function dummy parameter". If an error in the parameter list is detected, recovery is made to the arithmetic expression on the right of the equal sign.

The arithmetic expression corresponding to the statement function definition is processed. Note that the expression may reference both variables of the module as well as dummy parameters. FSTFUN classifies all members of the arithmetic expression as simple variables; the symbol ambiguity process is required to distinguish program variables fromm dummy parameter references.

Special Note: The potential for error exists for function dummy parameters with the same character string as program variables. The ANSI Standard indicates these are independent quantities; FACES will not make the proper distinction if the variable is referenced first (before the statement function definition in source code sequence). The strategy for dummy parameter handling should be modified.

SUBROUTINE FSUB

Mnemonic Origin: FORTRAN SUBROUTINE definition.

Classification: FFE statement processor.

Purpose: Process SUBROUTINE statements.

Operation: The subroutine name is extracted from the run on character string SUBROUTINE. The subroutine name is assigned as the current module name and the module type set to "Subroutine". The name is recorded in the symbol table and the initial entry transition is recorded. If a parameter list is present for the subroutine, the list is processed.

SUBROUTINE FTYPE (KT, HDRCNT)

Mnemonic Origin: FORTRAN Type Statement.

Classification: FFE statement processor.

Purpose: Process statement forms which begin with a type declaration.

Operation: Upon entry, a statement has been recognized which begins with a type declaration. The statement may be a simple type statement or a function which is preceeded by an explicit type declaration.

By parameter, the type identifier and length of the identifier is indicated by the calling routine. The type identifying characters are skipped and the characters which follow the type are examined. If the character string "FUNCTION(" is found, the statement is a function declaration.

For function declarations, the function name is extracted and aligned on the first character. The function name and type designation are passed to the function statement processor.

If a function is not defined, the statement is a normal type declaration statement. The first variable is extracted and aligned on the first character of the variable name. The aligned name is placed in the Parsing Tables replacing current entries to normalize the declaration list appearance.

If the variable declaration process detects a /, control is returned. A DATA initialization assignment is recognized. To treat this form, the variable being initialized must be recorded with a DATA initialization use. This requires first positioning the symbol table back to the variable.

If the initial variable was unsubscripted, the symbol table is already at the correct position. If, however, the variable was an array declaration, the Parsing Tables must be moved back to the array name position. This is performed by searching the Parsing Tables backwards looking for the last left parenthesis.

After recovery to the array name, the symbol is rerecorded to position the Symbol Table to the proper location.

The use as a Data initialized variable is recorded and processing of the declaration list resumes.

Parameters: KT - type code corresponding to the type indicated by the type identifier.

HDRCNT - number of characters in type identification.

SUBROUTINE FUNCI (X, KT)

Mnemonic Origin: FORTRAN FUNCTION declaration.

Classification: FFE statement processor.

Purpose: Process Function declaration statement.

Operation: Entry to FUNCI may be gained either from the keyword process or via an explicit type declaration of the name. If the entry results from type processing, an explicit type declaration for the function is passed by parameter. If the entry is via the keyword process, the type is defaulted to the leading character of the function name.

The current statement type is set to a function declaration to override any previous assignment which might have resulted from leading type declaration. The current module name is assigned the function name. The state of the current module is examined; if the current module is already in progress, the header card indicates the absence of an END card on the current module. If this is the first statement of the module, processing proceeds.

Processing the function definition involves recording the function name in the symbol table and the initial entry transition. The Use of the function name is recorded as a declaration. The parameter list is then processed recording dummy parameters in the Symbol and Use tables.

Parameters: X - Function name in 2A4 format, extracted prior to calling FUNCI.

KT - type specification for function name. Contains a type code if explicit type specified and the value zero otherwise.

SUBROUTINE FUNPAR

Mneumonic Origin: Function Parameters

Classification: AIR Query

Purpose: Searches for function parameters assigned values within function itself.

Operations: Warning flags may be produced for primary listing only. Program boundaries are not crossed. It is assumed that external references within function do not modify functions parameters.

Algorithm: See Source Code Listing.

SUBROUTINE FUNREF (CLASS)

Mnemonic Origin: Function Reference.

Classification: FFE parsing support routine.

Purpose: Generate table entries for a function reference.

Operation: Upon entry, the Parsing Tables are positioned to a function (external or statement function) which is being referenced. The distinction between statement functions and external functions is made by examining the parameter passed. The function name and Use are recorded followed by a call transition. The function actual parameters are then processed. By assumption, the function actual parameter list has been reduced previously. Thus, entries are either simple operands or array references with simple subscripts.

Control is returned with the Parsing Tables positioned to the entry following the right parenthesis of the parameter list.

Parameters: CLASS - contains a class code for the function name. Either a statement function or external function.

SUBROUTINE FVARDC (USECOD, TYP)

Mnemonic Origin: FORTRAN Variable Declarations.

Classification: FFE parsing support routine.

Purpose: Process declaration of variables.

Operation: Upon entry, the Parsing tables are positioned to a comma separated list of variables being declared by TYPE, COMMON, or DIMENSION statements. If a subscripted variable is present, an array declaration is assumed.

FVARDC accommodates two types of declarations: those requiring Use code recording for appearance in the statement (e.g., COMMON) and those not requiring Use codes for appearance in the statement (e.g., DIMENSION). If the variable requires recording statement appearance, the calling routine passes a positive (i.e., nonzero) Use value by parameter. If Uses are not required, the calling routine passes a zero valued Use specification.

If no Uses are required, FVARDC scans the Parsing table entries for the next array declaration. If the list is exhausted, control returns to the calling routine. If an array declaration is found, the array is processed.

If statement appearance Uses are required, variable entries are recorded using the Use code provided by parameter.

Whether statement appearance Uses are required or not, an array declaration causes the Use table description of declared array dimensionality to be recorded. The array name symbol must be changed from the initial recording of "scalar variable" to "array" in the Symbol table. Following array declaration Use a series of dimensions are recorded.

Processing terminates when a statement entry other than a variable or declared array is detected in the Parsing tables.

Parameters: USECOD - use specification. If positive, the Use code to assign to statement entries. If zero, no uses are recorded except for declared arrays.

TYP - type specification for entries of the declaration. Maybe forced to a particular type by calling routine, or defaulted to type implied by variable name (zero value).

SUBROUTINE FWRITE

Mnemonic Origin: FORTRAN Write statement.

Classification: FFE statement processor.

Purpose: Process WRITE forms of statements.

Operation: Statement forms processed are the keyword WRITE followed by an I/O control list and optional I/O variable list. Any other form is considered a target machine dependent form and not processed.

SUBROUTINE GENTEM (TYPE, NAME)

Mnemonic Origin: Generate temporary name.

Classification: FFE table production support routine.

Purpose: Produce a nonambiguous name for replacing a structure by a temporary.

Operation: Upon entry, a nonambiguous name is required for assignment of a substructure (e.g., the expression used as an actual parameter to a subroutine). The type code of the resulting temporary is passed to permit maintenance recognition of the temporary type generated.

The temporary is generated in two parts: the first integer word is assigned a character string which cannot be a valid FORTRAN variable name to avoid conflict with user selected names. The second word is assigned a pattern which uniquely identifies the temporary within the statement. After name generation, the second word name generator is advanced to insure uniqueness of the next name. Reset of the name generator is performed by some external routine.

Parameters: TYPE - Type code for name to be generated.
NAME - return parameter for generated name in 2A4 format.

Special Notes: The character string generated by the current incrementation process may produce special characters or unprintable bit patterns. This is not signification since temporary names always have a nondefault type associated.

The first word of the temporary name is selected from a list of templates set in BLOCK DATA. The leading character indicates the type code specified when the name was generated.

For protection, the leading characters are assigned to minimize error potential in default typing if the type is erroneously lost during processing.

SUBROUTINE GETCLS (CURCLS)

Mnemonic Origin: Get Class code of current symbol.

Classification: FFE table searching routine.

Purpose: Indicate the class code assigned to the currently selected Symbol table entry.

Operation: Upon entry, the Symbol Table is positioned to an entry for which the class must be investigated. Normally, this interrogation is from a parsing routine which must distinguish processing cases through the class code. To prevent unnecessary access to the Symbol Table, this routine interfaces the Symbol Table structure to the processing routine.

If the Symbol Table is not in an overflow state, (i.e., there is a nonempty symbol selected), the class code of the symbol is returned through the parameter. If the Symbol Table is in an overflow state, no symbol is selected and the value 0 is returned.

Parameters: CURCLS - return parameter for extracted class code.

SUBROUTINE GETE(TAB, LC)

Mnemonic Origin: Get Element

Classification: AIR General Purpose Utility

Purpose: Allows access to values of elements in permanent AIR tables. (local and global tables)

Operation: Algorithm: Binary tree search through permanent table names, followed by fetching element and placing it into Element Register.

Parameters: TAB - Input

LC - Input

Column Register (CR) is set to logical column LC. Element in table TAB at logical column LC and at row indicated by current row pointer to table TAB, along with associated information, is placed into Element Register (ER). (see "AIR Abbreviations").

see also /SPEREG/

SUBROUTINE GETFLG

Mnemonic Origin: Get Flag

Classification: Report Generator message construction routine

Purpose: Acquire the next flag from the Flag File and suppress redundant adjacent flags on the file.

Operation: Upon entry, the last flag returned occupies the next flag COMMON data. (This flag is initialized to a neutral configuration for the first call). The next nonredundant flag is to be returned to the calling routine.

The next flag description is set empty. If the current flag buffer is empty a new flag is requested from the I/O service routine. The flag buffer is then compared to the old next flag contents. If they are the same, a redundant flag has appeared; redundant data is discarded by simply setting the current flag buffer empty and repeating the cycle.

When a nonduplicate is found, new data is transferred to the next flag description marking the buffer full. If an end of file occurs on the Flag File, the I/O service routine will return an empty flag in the buffer. This action will terminate the search loop since the next flag marker is set by the flag buffer variable.

End of file causes a neutral flag to be returned to the calling routine advancing the global key. Global key advance allows flag exhaustion to occur on a report boundary.

SUBROUTINE GETMSG

Mnemonic Origin: Get Message

Classification: Report Generator message construction routine.

Purpose: Construct the next message from flags, suppressing redundant messages.

Operation: Upon entry, the next message is required from the Flag File entries. The message buffer is occupied by the last message transferred to report generating routines; the message is initialized to a neutral state for the first call. Flags are acquired and combined into a set of data constituting a message.

By convention, report processing routines are to confirm the consumption and use of messages as they are passed. The status of the last message is examined prior to constructing the next message. If the pointers indicate the last message was not properly processed, a warning is issued.

The current message length is set empty and the next flag acquired if the next flag buffer is empty. A loop is then entered to construct the message from a series of flags.

If the first flag of the message is empty, no more flags are available. A neutral message is constructed with empty length and a key value greater than the key of the last message. This will cause the end of messages to occur at the beginning of a new report, yet allow any report current in progress to complete.

If the first flag is nonempty, operation is dependent upon the characteristics of the next flag. If a redundant message is being constructed, the data contents of the message being built will be the same as the last message constructed. If message identification (i.e. key value, relative card indicator, flag indicator, and order indicators) are the same as the last message, a potentially redundant message is recognized. Notice that the data will be identical for a redundant message; therefore, no values are inserted in the message descriptors. Insertion in the message descriptors is only performed if a different message is beginning.

After setting the message characteristics, flag data is appended to the message until a flag with different characteristics is found. To accomplish this function, data insertion is performed only if the flag contains different data from the information currently resident in the message buffer. If new data is encountered, the information is physically inserted in the buffer; if identical data is detected, the current information is allowed to remain undisturbed.

After processing data fields, the next flag is requested. If the next flag contains an identification which differs from the message identification established by the first flag, insertion is halted. Insertion is halted also if the message data buffer is filled.

Notice that the data inserted in the first part of the loop is derived from the initial flag on the first iteration, then from the flag retrieved at the bottom of the loop on subsequent iterations. This permits message construction to look ahead at the incoming flags and accept data only if it belongs to the current message.

After constructing the message, the results of the process are examined. If a redundant message has been constructed, the message is discarded and the process begins anew. Otherwise, the length of the current message is recorded in a saving variable for use on the next message. This length will be used to distinguish valid data in the message from left over entries in the comparison.

The pointer to the message text is set to the first entry of the message to initialize text processing by report generating routines.

Special Notes: The need to suppress redundant messages required several unusual techniques. Since the data of the last message is used to detect redundance, message processing routines should not modify the contents of the message. Modification of information may defeat the redundancy mechanism resulting in duplicate messages or undesired suppression of the next message.

Since message processing routines report the processing of messages by setting the message length empty, the area occupied

by data must be saved in the special COMMON variable LASTPL. This value is used from one call to the next in redundancy searches. The value should not be accepted (except in initialization) by any other routine.

Reusing duplicate information rather than inserting the Flag information may cause some confusion. The transfer of multiple items did not seem worthwhile since this would be totally duplicate effort.

Notice that redundant messages are not necessarily identical. Superficial differences are permitted in the message description fields. Additionally, if a second message is a proper subset of the previous message, the message is suppressed as redundant.

SUBROUTINE GETL(BF)

Mnemonic Origin: Get Local Module

Classification: AIR General Purpose Utility

Purpose: Bring local module into main memory.

Operations: If Forward-Backward Register (FBR) indicates "forward", bring local module indicated by Element Register (ER) into main memory. If FBR indicates "backward", bring local module indicated by top of Control Stack into main memory.

After requested module has been brought into main memory, BF is set to one; if it is not brought in, BF is set to two.

Module Register (MR) is set to module number of module brought in.

Parameters: BF - Output

See also /SPEREG/

SUBROUTINE GETSCA(SCAL, TAB, VALUE)

Mnemonic Origin: Get Scalar

Classification: AIR General Purpose Utility

Purpose: Allows access to values of scalars associated with permanent AIR data structures.

Operation: Algorithm: Binary tree search through permanent data structure names, followed by binary tree search through scalar identifier names (see "AIR Abbreviations"). VALUE is then set.

Scalars whose value may be accessed:

1. length of data structure (table or stack)
2. current row pointer to data structure
3. pointer to last non-empty (valid) row in data structure
4. prime number associated with hash-coded table.

Parameters: SCAL - Input

TAB - Input

VALUE - Output

VALUE is set to value of scalar indicated by SCAL associated with data structure TAB.

see also SETSCA

SUBROUTINE GETTST (OARY, OLNG, PTR, OVER)

Mnemonic Origin: Get Temporary Symbol table entry.

Classification: FFE table manipulation routine.

Purpose: Extract the contents of the currently addressed Temporary Symbol Table entry and unpack the character string into the provided array.

Operation: Upon entry, the currently selected Temporary Symbol Table entry is required to construct an alternate form. The contents of the entry are to be extracted and unpacked. Since the actual contents of the symbol contain filler blanks inserted on the right of the symbol character, unpacking need only proceed to the first blank character.

Accessing the symbol string requires interpretation of the currently selected symbol. If the character string is less than 8 characters, the character string is contained in the main Temporary Symbol Table entry; if more than 8 characters are involved in the symbol, the character string is contained in the overflow table of the Temporary Symbol Table.

Once the character string is extracted, the characters are unpacked and placed in the output vector until the character string is spanned, a blank character is detected, or the output vector space exhausted. If the output vector is filled before the character is completely extracted, processing is terminated and the overflow indicator is set for calling routine interpretation.

Parameters: OARY - output vector for receiving unpacked characters in A1 format.

OLNG - length of output vector available for receiving characters.

PTR - insertion pointer for placing characters in the ouput vector. Advanced with each character inserted.

OVER - output indicator set to inform calling process that the vector space was exhausted before the character string was fully unpacked.

SUBROUTINE GETTYP (CURTYP)

Mnemonic Origin: Get Type code of current symbol.

Classification: FFE table manipulation routine.

Purpose: Extract the Type code of the currently addressed Symbol Table entry.

Operation: Upon entry, a processing routine requires inspection of the Type code assigned to the currently selected Symbol Table position. This routine interfaces Symbol Table construction to other routines to avoid proliferation of the Symbol Table.

If the Symbol Table is not in an overflow state, the Type code of the currently selected entry is extracted and returned to the calling routine. If the Symbol Table is in overflow (no valid current symbol), the value 0 is returned.

Parameters: CURTYP - return parameter of the Type code assigned to the current symbol.

FUNCTION HASHSY(NAME1, ANME2, CLASS, N)

Mnemonic Origin: Hash into Symbol Table

Classification: AIR General Purpose Utility

Purpose: Hash to location in Symbol Table.

Operations: Hash into Symbol Table, searching for symbol string contained in NAME1 and NAME2, and having any class code contained in array CLASS (which has N entries).

If symbol string located with matching class code, HASHSY set to location of symbol string in Symbol Table. Else, HASHSY set to zero.

Parameters:

NAME1	-	Input
NAME2	-	"
CLASS	-	"
N	-	"
HASHSY	-	Output

INTEGER FUNCTION HIFECH (IWORD)

Mnemonic Origin: High Fetch of half word data.

Classification: System utility.

Purpose: Extract the upper half word of data from a full word integer.

Operation: A full integer data word is provided by parameter. The upper half word of the data is extracted and right justified with zero left fill. This information is returned to the calling routine via the function name.

Parameters: IWORD - integer full word of data from which the upper half word is to be extracted.

HIFECH - right justified half word returned through the function name.

INTEGER FUNCTION HISTOR (LHALF, UHALF)

Mnemonic Origin: High Store.

Classification: System bit manipulation utility.

Purpose: Construct a packed full word integer from two lower half words of data.

Operation: A full word integer is constructed from the least significant bit positions of two integers. The resulting full word is composed of packed data of the lower half words.

Parameters: LHALF - input parameter containing data in the LSB half word. This data becomes the low half word of the result.

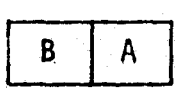
UHALF - input parameter containing data in the LSB half word. This data becomes the high half word of the result.

HISTOR - the constructed integer is returned through the function name.

Special Note: The parameters of HISTOR are intuitively reversed. The original concept was the insertion of a low order half word into the higher order half word bits of the other parameters. For example in the call,

W = HISTOR (A, B)

the configuration returned to W is an integer construction:



where B has been inserted into the upper half of A.

SUBROUTINE IE(TAB, BF)

Mnemonic Origin: Initial Entry into Table

Classification: AIR General Purpose Utility

Purpose: Allows access to entire tables, from first to last entry.

Operation: Algorithm: IE performs one of two complex operations, depending on whether Forward-Backward Register (FBR) indicates forward or backward.

Forward - Find initial entry point into table TAB, (i.e., first row in table). Place this information and associated information into top of Control Stack.

Backward - Find next row in table. Update information at top of Control Stack.

Parameters: TAB - Input

BF - Output

Traverse table TAB, one row at a time. If table entirely traversed, branching flag BF is set to two; else, BF is set to one.

Table IE accesses: COM, DIR, IS, NOD, SH, SYM, USE1, USE2. (See "AIR Abbreviations")

See also TT, "AIR Basic Search Technique", "Pattern Searches", and "Traversing Lists".

SUBROUTINE IMPLDO(IFLAG)

Mnemonic Origin: Implied DO Loop Index Variable Search

Classification: AIR Special Purpose Utility
(Referenced only during search for uninitialized local variables and for DO loop index variables used after loop has terminated normally).

Purpose: Determines if variable is set by implied DO loop.

Operation: IFLAG indicates whether or not variable indicated by current row pointer to Statement Number Linked Use Table (USE1) is set by implied DO loop.

Algorithm: See Source Code Listing.

Parameters: IFLAG - Output

Designer's Comment: This routine exists only because an implied DO is not treated as a normal DO loop. This problem should be rectified in the future.

SUBROUTINE INxxx

Mneumonic Origin: Input xxx (Global) Tables, where

xxx = COM, ISH, SH

Classification: AIR System Request Routine

Purpose: Read tables xxx into main memory from secondary storage.

Operations: COM: COMMON Block Reference Table and Linked List Table

ISH: Inverse System Hierarchy Table and Inverse System Hierarchy to Directory Table

SH: System Hierarchy Table and System Hierarchy to Directory Table.

SUBROUTINE INDIR

Mnemonic Origin: Input DIRECTORY.

Classification: System I/O routine from Table File.

Purpose: Acquiring Directory from the Table File to initialize system to last status.

Operation: The Directory contents are read from the Table File containing Directory records. The pointer to the last Directory entry is acquired from the global header. To guard against invalid data from incorrect file attachment, the pointer information is checked for consistency. If incorrect data values are found, the Directory is set empty and a warning message issued.

Upon return, the Directory has been initialized to a state usable by other system components.

SUBROUTINE INGHD

Mnemonic Origin: Input Global Header.

Classification: System I/O routine.

Purpose: Read contents of Global Header from the Table File.

Operation: The Global Header contents are acquired from the Table file. Contents of the Global Header are not checked; this verification is performed elsewhere.

SUBROUTINE INTAIR

Mnemonic Origin: Initialize AIR

Classification: AIR routine

Purpose: Initialize AIR subsystem.

Operation: Set lengths of data structures. Initialize special purpose registers.

SUBROUTINE INTFFE

Mnemonic Origin: Initialize FORTRAN Front End.

Classification: FFE initialization routine.

Purpose: Initialize FFE data constructions which are not needed by other subsystems.

Operation: INTFFE acts as a surrogate BLOCK DATA for the FFE. The purpose of INTFFE rather than BLOCK DATA is to isolate data structure initialization from other FACES system processors enabling overlays to exclude COMMON data needed by FFE from other overlays.

The length of local table and processing tables are established. The initial values of I/O buffers and error message reporting data structures are initialized. An empty statement is inserted in the Parsing tables to initiate processing of FORTRAN text.

SUBROUTINE INTRPT

Mnemonic Origin: Initialize Report Generator

Classification: Report Generator initialization

Purpose: Perform initialization for report generator data which is not needed in all other subsystems.

Operation: Lengths and initial values of Report Generator data structures are established. Initial data accommodates the initial transient of I/O routines and duplicate data suppression routines.

INTEGER FUNCTION ISSCLS (ISSCOD)

Mnemonic Origin: Intermediate Symbol String Class code.

Classification: FFE table construction support function.

Purpose: Assign a Class code for a Symbol Table entry based upon the ISS code of the entry.

Operation: Upon entry, an ISS code is presented for which an implied Class code is desired. If the presented symbol can be classed by examination of ISS code, the Class code is returned. If the symbol cannot be classed, the value 0 is returned.

Cross Reference: See description of Parsing Tables (ISS).

Parameters: ISSCOD - ISS code of the entry needing class assignment.
ISSCLS - Assigned Class code returned through function name.

SUBROUTINE ISSTYP (ISSCOD)

Mnemonic Origin: Intermediate Symbol String Type code.

Classification: FFE table generating support routine.

Purpose: Assign a Type code based upon the ISS coded entry.

Operation: Upon entry, an ISS coded entry is presented for which the Symbol Table Type code is needed. If the Type code can be determined from the ISS code of the entry, the Type code is returned. If the type cannot be determined from the ISS code, the value 0 is returned.

Parameters: ISSCOD - Intermediate Symbol String entry code.
ISSTYP - Type code assignment is returned through the function name.

Cross Reference: See Parsing Table description for ISS Type codes implied from ISS entries.

SUBROUTINE ISYSTM (RERUN)

Mnemonic Origin: Initialize system.

Classification: Control Driver initialization routine.

Purpose: Initialize the FACES system by either acquiring system status for existing table file data (rerun) or constructing an initially empty system (initial run).

Operation: Upon entry, the FACES system is about to begin operation. The system is to be initialized from the current state of the software under analysis, or set empty to receive the first set of modules for a new software system. The passed control parameter is examined to determine which activity to perform.

If an initial system is to be created, a Global Header is constructed. The source catalogue file is emptied and the Directory set empty.

If a rerun is required, the system is initialized from Table File data. The Global Header is acquired from the Table File and verified. If fatal errors are detected in the Global Header, the run is aborted. This may be caused by an invalid file being passed as the Table File to the system.

If a valid header is read, the contents of the current Directory are transferred from the Table File to the core resident COMMON block.

Parameters: RERUN - input control parameter indicating whether the system is to be initialized for the first run or linked to a system already created.

SUBROUTINE KEYWDM (WORD, INDEX)

Menmonic Origin: Key Word Match.

Classification: FFE parsing support routine.

Purpose: Identify the presence of a FORTRAN Keyword.

Operation: Upon entry, KEYWDM is provided with the leading four characters of a possible FORTRAN keyword. If this character string matches the leading four characters of an established keyword, a positive index value is returned. If no match is found, the value zero is returned as the index.

KEYWDM uses a read-only table of keywords which are loaded through BLOCK DATA. Access to the table entries is made by using the third and fourth characters of the presented character string (see description of Keyword Match table, MATCH, for access method).

If a table entry is found which matches the presented character string, the value of the table address is returned through the index parameter. If the addressing method produces an invalid table address, the no match condition is detected and a zero index is returned.

If the table search discovers an empty table position the search is halted and the no match condition is returned.

Parameters: WORD - leading four characters of the suspected keyword in A4 format.

INDEX - returned value of the table position matched or the value 0 if no match found.

Special Notes: Value of the match pointer is not significant from one search to another. This value is set equal to the return value for the index to assist in maintenance tracing activity.

SUBROUTINE LINEPR(ORIGIN, FIRSTC, LASTC)

Mnemonic Origin: Line print

Classification: Report Generator service routine

Purpose: Print a series of source code lines for a report.

Operation: Upon entry, a (possibly empty) set of cards are to be printed for a Primary or Secondary report. The card set is described by a module origin in the Source Code Catalogue and relative card numbers of the source to be printed.

If the card set description indicates an empty set, control is returned to the calling routine without printing any source lines. If a nonempty set is described, the card set is printed.

The Source Code Catalogue is positioned to the first card image of the set and the cards are printed in a loop. The number of card images are computed and the card number of the first card established. In addition to the actual source code image, an identification of the card is printed to assist location of the card in the sequence. The primary entry point name is obtained from the current module description.

Parameters: ORIGIN - Source Code Catalogue origin of the card image set.

FIRSTC - relative card number of the first card in the set to be printed.

LASTC - relative card number of the last card in the set to be printed.

Special Notes: Source code catalogue positioning is a protective mechanism for Primary Reports. For Secondary reports, this positioning is required for proper operation.

SUBROUTINE LIRL(LIST)

Mnemonic Origin: Load Immediate Register from List

Classification: AIR General Purpose Utility

Purpose: Access information in lists of List Table.

Operation: Load Immediate Register array with List Table element indicated by current pointer of list LIST in List Table Map.

First element in Immediate Register is set to type (alphanumeric or integer) of List Table element. Second element contains width, in computer words, of List Table element. List Table element itself is placed in remaining elements of Immediate Register, left-adjusted.

Parameters: LIST - Input

See also /LIS/, MANL, and List Table.

SUBROUTINE LNKAIR

Mnemonic Origin: Link Automatic Interrogation Routine

Classification: Control Driver subsystem linkage routine

Purpose: Position files, set controls, and activate the Automatic Interrogation subsystem.

Operation: Upon entry, a QUERY command card has been recognized. The Analysis Table File contains module descriptions to be investigated for the features specified on the QUERY card. QUERY options are interpreted to set controls for the AIR subsystem.

Files are positioned for the start of the AIR subsystem. Read only data files are rewound and the Flag File is positioned to the end. AIR flags will be appended to the end of the Flag File.

Query options are interpreted and the selected queries listed for user information. AIR is then activated. Upon return, AIR flags have been produced for the requested analysis and global tables (if necessary) constructed for Analysis Table Files. The Flag File is marked before return to seal information generated for subsequent reports.

SUBROUTINE LNKFFE

Mnemonic Origin: Link to FORTRAN Front End.

Classification: Control Driver subsystem link routine.

Purpose: Position files and initiate activities for FFE sybsystem execution.

Operation: Upon entry, a user request to add FORTRAN modules to the software system under analysis has been recognized. The Source Code Catalogue is moved to the end of current FORTRAN text recorded. The Flag File is positioned to the end of current recorded Flags (possibly empty). The FFE is activated to incorporate new FORTRAN modules in the software system.

After addition to the software system, global tables recording COMMON block usage and Calls among routines are incomplete. The last entries to these tables is reset to zero to force their recreation.

The new end of Source Code Catalogue and Flag File entries is recorded and control returnsto the calling routine.

SUBROUTINE LNKRPT

Mnemonic Origin: Link Report Generator

Classification: Control Driver subsystem link process

Purpose: Position files and interpret command for report generation.

Operation: Upon entry, a REPORT command has been recognized. Report options are to be examined and the requested reports produced from Flag File information.

The Flag File is rewound for processing reports. At this point in the processing, the flags have been sorted in ascending order on the Flag File.

Remaining entries (if any) on the REPORT card are examined to set control variables for report generation. FLAGed reports are defaulted unless an ALL report is requested. The Report Generator is activated to produce the reports. Upon return, the Flag File emptied since results have been reported to the user.

SUBROUTINE LOCPRT

Mnemonic Origin: Local Table Print.

Classification: FFE maintenance support routine.

Purpose: Display the local tables of a module on a print display.

Operation: Upon entry, the local tables are occupied with data belonging to the module indicated in /CURMOD/. A header line indicating the module is printed using values from /CURMOD/ followed by the table data. Maintenance dump routines are used to actually perform the table printing.

INTEGER FUNCTION LOFECH (IWORD)

Mnemonic Origin: Low Fetch

Classification: System bit manipulation utility.

Purpose: Extract the lower half word from a full word.

Operation: The LSB half word is extracted from a full word integer. The resulting half word is returned with zero left fill in the upper half word bit positions.

Parameters: IWORD - full word of integer data containing two half word fields.

LOFECH - extracted half word returned through function name.

INTEGER FUNCTION LOSTOR (UHALF, LHALF)

Mnemonic Origin: Low Store

Classification: System Bit manipulation utility.

Purpose: Construct a full word integer with two half word fields from the upper half of one word and lower half of another word.

Operation: The lower half word of an integer is cleared to zeroes and the upper half word of the other integer is cleared to zero. Results from the clearing operation are combined to create a full word integer with two half word fields.

Parameters: UHALF - integer word containing data in the upper half word.

LHALF - integer word containing data in the lower half word.

LOSTOR - the constructed full word integer is returned through the function name.

SUBROUTINE LSTLNK (USECOD)

Mnemonic origin: List link.

Classification: FFE table generating routine.

Purpose: Create Use table linkage among Begin/End list bracket entries.

Operation: Upon entry, a Use Table entry is created which is either a Begin or End List Bracket. Forward and backward pointer entries are required for the Use entry.

Begin and End Use codes are treated separately. A single routine was implemented to ease the understanding of operation. Begin and End entries are linked to one another on separate calls, using the List Bracket Stack to record the last occurrance of Begin Use codes.

If the Use code is a Begin bracket, the pointer to the Symbol Table is examined. If the Symbol Table is positioned to a nonempty entry a dependent list is required linked to the selected symbol. If the Symbol Table is in an overflow state or positioned to an empty entry, an independent list is required.

Note that construction of dependent lists assumes that at least one Use of the symbol has already occurred; therefore, a Begin list bracket cannot point directly back to the Symbol Table entry. The Begin list bracket points back to the most recent Use of the selected symbol.

On Begin List Use code, the backpointer is inserted in the current Use Table position and the Use Table position of this Begin code is inserted in the List Bracket Stack. The forward pointer of the Begin Use entry is not modified during this insertion.

When an End List Use code is encountered on a later call, the Use Table position of the most recent Begin code is retrieved from the List

Bracket Stack. The forward pointer of the Begin Bracket Use code is set to the current Use Table position and the backpointer of the current Use Table entry is set to the Begin pointer position. In this way, the two Bracket codes are linked together. The forward pointer of the End Use code is set to zero. Note that recording list links for End list code is the same whether the list is dependent or independent.

Parameters: USECOD - Use code indicating whether a Begin or End list code linkage is to be generated.

SUBROUTINE MODNAM(NUM, N1, N2)

Mnemonic Origin: Module Name Search

Classification: AIR General Purpose Utility

Purpose: Find name of module having module number NUM when AIR suffers terminal error.

Operation: Place name of module having module number NUM in N1 and N2, four characters per word, left-adjusted. If name is not found, fill N1 and N2 with asterisks.

Parameters: NUM - Input

N1 - Output

N2 - Output

SUBROUTINE MULBRA

Mneumonic Origin: Multiple Branch Statements

Classification: AIR Query

Purpose: Searches for multiple branching statements not branching to statement immediately following

Operations: Warning flags may be produced for primary listing only.

Algorithm: See Source Code Listing

SUBROUTINE MVMSG(REPORT, OROGIN, CARD)

Mnemonic Origin: Move Messages

Classification: Report Generator service routine

Purpose: Position messages to the first message for the indicated report.

Operation: Upon entry, a report is about to begin. The first message is to be constructed from Flag File contents. The Flag File, a sequential medium, may already be positioned to the proper location, or contain information not properly processed on the last report. MVMSG recovers from report processing errors and skips incorrect flags which might otherwise influence processing.

If the current message is empty, or exhausted, a new message is requested. If the message is empty, (i.e. no message produced by the request), Flag File contents are exhausted. This condition satisifies the movement requirement.

If the message is not empty, but a message is present which is behind the requested message, messages are discarded until a message is found which satisifies the required conditions, or the end of Flags are encountered.

Parameters: REPORT - integer input specification indicating the Flag key field for an acceptable message.

ORIGIN - source code origin specification for an an acceptable message.

CARD - relative card number specification for an acceptable message.

Special Notes: Current implementation of MVMSG requires requests for messages to appear in the same order as the Flag File contents are sorted. That is, a sequential Flag File and report order request are assumed. If reports are requested in random order, valid messages will be skipped where the sequence of requests differs from the Flag File sort order.

SUBROUTINE MVSCAT(ORIGIN, RECORD)

Mnemonic Origin: Move Source Code Catalogue

Classification: System I/O positioning routine

Purpose: Position the Source Code Catalogue to the indicated record position.

Operation: Upon entry, it is necessary to position the Source Code Catalogue to a given physical record position. The indicated record position will be the next record to be read or written.

The physical record number is computed using the origin and relative card number. If the record is within Source Code Catalogue bounds, the file is positioned to the requested location.

Parameters: ORIGIN - Source Code Catalogue origin.

RECORD - Relative card number within specified origin.

INTEGER FUNCTION NEEDTP (NAME, LNG, CLASS)

Mnemonic Origin: Need Type.

Classification: FFE Table generation routine.

Purpose: Provide a Type code for defaulted Symbols when inserting Symbol Table entries.

Operation: A symbol is presented for which a type is required before inserting the symbol in the Symbol Table. NEEDTP has both active and error protection code for the typing of symbols.

Normally, NEEDTYP processes symbols which are variable and function names not previously assigned types. These entities are discovered by examining the class code assigned. The symbols are typed by examining the first character of the symbol name.

Additionally, NEEDTYP provides type codes for alphabetic character strings used in a constant context (e.g., DATA constants).

Other operations are error neutralizing activities used to cover the possible omission of typing other processing routines. If an invalid class code is discovered, the type of zero is returned.

Parameters:

- NAME - Symbol name of symbol in A4 format.
- LNG - Length of symbolic name in number of integer words.
- CLASS - Class code assigned to symbolic name.
- NEEDTYP - Assigned type code returned through function name.

SUBROUTINE NEWDIR

Mnemonic Origin: New Directory.

Classification: FFE Table Generation Routine.

Purpose: Create new Directory entries from symbol references of module processed.

Operation: Upon entry, a module has been processed. The tables generated for the module are used to update the Directory contents. The primary entry point to the module is contained in the current module description, /CURMOD/. The module type is used to determine the Class code of the module name found in the Symbol Table. The Symbol Table is then searched for the module name (primary entry point) to begin the investigation for Directory entries.

The primary entry point is recorded in the Directory along with access information to the Table File entries and Source Code Catalogue index information for module source. The other active symbols of the module are then searched for recordable Directory entries.

Other recordable Directory entries are secondary entry points and references to subroutines and functions. If a secondary entry point is found, the entry is recorded with the same file access information as the primary entry point. References to other routines are recorded in the Directory with empty access information since these are references, not definitions, of the modules.

If an EXTERNAL function or subroutine is found, the character string may not be the name of the routine. For example, if the routine name is passed by parameter, the name used in the CALL may not be the routine name. Therefore, references to external declared routines are

entered in the Directory only if they appear in an EXTERNAL statement. Since reference entries of the Directory are overridden by definition of the module, the reference to an EXTERNAL routine is recorded in the Directory as a subroutine; if it turns out to be a function later, or already is defined as a function, the function declaration will correct this error.

SUBROUTINE NODGEN

Mnemonic Origin: Node Generation.

Classification: FFE Table Generation Routine.

Purpose: Manage the production of graphical flow of control for a module.

Operation: NODGEN provides a processing sequence required to generate graphical nodes of the module just processed. The transition pairs table is first converted to replace references to symbolic labels with references to node (statement) numbers. Then the program graph successors are produced, followed by graphical predecessors.

SUBROUTINE NOPRO

Mnemonic Origin: No Process Statement.

Classification: FFE Parsing Support Routine.

Purpose: Provide artificial parsing of statements not processed by FACES.

Operation: Upon entry, a FORTRAN statement has been recognized which is not processed by current code. A warning flag is issued to inform the user that the statement text will not be included in analysis. The Parsing Tables are positioned to the end of the recorded entries to force completion of the statement. No entries are made in the Symbol or Use tables for the statement.

Special Note: NOPRO is not called when ignoring a statement will not influence the analysis. For example, ignoring a FORMAT statement text will not influence any current analysis.

SUBROUTINE NORLNK

Mnemonic Origin: Normal Link.

Classification: FFE Table Generation Routine.

Purpose: Provide normal Use Table linkage among entries.

Operation: Upon entry, a normally linked Use code is to be recorded in the selected position of the Use Table. This Use code is not a special bracket code. The Symbol Table is positioned to the symbol for which the Use is to be recorded.

If the Symbol Table is in an overflow state or the Symbol Table position is empty, a warning message is issued. Otherwise, the linkage of the Use entry to preceeding Uses of the symbol are recorded.

If this is the first Use of the symbol, the backpointer of the Use Table entry is set to a flagged half word pointing to the Symbol Table position. The first and the last Use pointers of the Symbol Table entry are set to the current (first and only) Use position.

If the symbol has been used previously, the backpointer of the new Use is set to the most recent Use, and the most recent Use forward pointer is set to the current Use Table position. The forward pointer for the current Use is set to zero since there is no Use in a forward position.

SUBROUTINE NXTCHR (KIND, OCHR)

Mnemonic Origin: Next Character.

Classification: FFE Scan Support Routine.

Purpose: Provide the next character to FFE scan routines from the Scan Buffer.

Operation: Upon entry, the next character is required from the Scan Buffer for statement text of a FORTRAN statement. The pointer to the Scan Buffer is advanced. If the Scan Buffer data has been exhausted, the buffer manager is called to provide more statement text.

A character is extracted from the Scan Buffer. If the request is for the next sequential character (e.g., in scanning Hollerith character strings), the extracted character is returned. If the request is for a nonblank character, the character is discarded if blank; blank characters are continually discarded until a nonblank symbol is found.

The end of statement is determined by the buffer manager. The buffer manager is responsible for inserting end of statement codes in the Scan Buffer when statement text is exhausted.

Parameters: KIND - input control parameter indicating the category of character sought.

OCHR - output parameter receiving the symbol in A format.

SUBROUTINE NXTCMD

Mnemonic Origin: Next Command.

Classification: Control Driver Command Interpretation.

Purpose: Provide the next command item from the command cards.

Operation: Upon entry, the next command item is required from the user's command card set. Several conditions may exist:

1. The first command is required from the first card.
2. A command card may currently be in progress.
3. A command card may be exhausted.
4. All command cards may have been processed.

If the first command from the first card is required, the command card image contains an empty command card (nonempty length pointer of zero value) and the command item is also empty; these values are established by initialization of the system.

If the current card is empty, a command card is requested from the command card set. If cards are available, the text image is returned. The command card is echo printed and the command card pointer is set to the first nonblank column. If a blank card is read, the pointer adjustment will cause an exhausted card to be indicated (i.e., the current pointer beyond the last nonempty pointer).

If nonblank characters remain on the command card, a command item is constructed from command card characters. Otherwise, the End of Card command item is returned as the next command item, and the command card image is set empty.

If, when requesting a new command card, no card is returned, the Finish of command cards is indicated. The Finish command is returned

as the next command item. Notice that any subsequent call will produce a Finish command item since the no cards will be available. This feature is intended to contain any sequence errors in which the Finish command is erroneously ignored by processing routines.

Allowing both an empty command card and exhausted command card state permits treatment of several boundary conditions. The primary function is to permit a last command item to be constructed and returned, followed by an End of Command item which does not physically occupy card columns. In addition, the command item constructed may occupy the last card column of the command card. Advancing past the pointer to the last entry of the command card will terminate construction of the command item. The last command item might also be a single symbol in the last card column. Therefore, the pointer to the command card must be permitted to exceed the last pointer to insure this case is properly processed.

LOGICAL FUNCTION OPERND (ISSCOD)

Mnemonic Origin: Operand.

Classification: FFE Support Routine.

Purpose: Identify Intermediate Symbol String entries corresponding to operands.

Operation: Upon entry, an ISS coded entry is presented which may be an operand (i.e., variable name or constant). If the entry presented corresponds to an operand, a TRUE value is returned through the function name. Nonoperand entries produce a FALSE value.

Any ISS code which is an alphabetic symbol other than Relational and Logical operators is an operand.

OPERND is used in parsing routines to select operands used in the FORTRAN text. OPERND is also used in manipulations on the Parsing Tables to detect ISS entries for which TSTAB character strings are recorded.

Parameters: ISSCOD - Intermediate symbol string code to be inspected as a possible operand.

OPERND - logical inspection result is returned through the Function name.

SUBROUTINE OUTxxx

Mneumonic Origin: Output xxx (Global) Tables where
xxx = COM, ISH, SH

Classification: AIR System Request Routine

Purpose: Write tables xxx out to secondary storage from main memory.

Operations: COM: COMMON Block Reference Table and Linked List Table

ISH: Inverse System Hierarchy Table and Inverse System Hierarchy to Directory Table

SH: System Hierarchy Table and System Hierarchy to Directory Table.

SUBROUTINE OUTDIR

Mnemonic Origin: Output Directory to Table File.

Classification: System I/O Routine.

Purpose: Place the contents of the Directory on the Table File.

Operation: The contents of the Directory are written to Table File records allocated for the Directory. The nonempty length of the Directory contents is placed in the Global Header position allocated for saving the Directory status.

SUBROUTINE OUTGHD

Mnemonic Origin: Output Global Header.

Classification: System I/O Routine.

Purpose: Write the contents of the Global Header to the Table File.

Operation: Contents of the Global Header are written to the Table File to secure the data generated during the run. Writing the Global Header updates Table File information and Source Code Catalogue data created during the run for future access.

INTEGER FUNCTION PAKCHR (VECT, COUNT)

Mnemonic Origin: Pack Characters.

Classification: FFE Character Manipulation Routine.

Purpose: Pack characters provided in A1 format into A4 format word.

Operation: Upon entry, a vector of A1 formatted characters is presented for packing. The length of the character string is between 1 and 4. The first n-1 characters are packed into an integer word, left justified with zero right fill. If n is less than 2, no characters are packed by this operation.

Finally, the last character is placed in the word using the trailing right fill from the last character as fill for the packed string.

The routine is coded in a more general form than the current use as an A4 packing routine to permit flexible use in expanded systems.

Parameters:

VECT	-	one dimensional vector containing A1 formatted character string to be packed.
COUNT	-	count of the number of characters to pack into the return word.
PAKCHR	-	packed character string returned through the function name.

SUBROUTINE PARAL (ARRAY, ARRSIZ)

Mneumonic Origin: Parameter List Alignment Check

Classification: AIR Query Driver

Purpose: Drives Parameter List Alignment Check

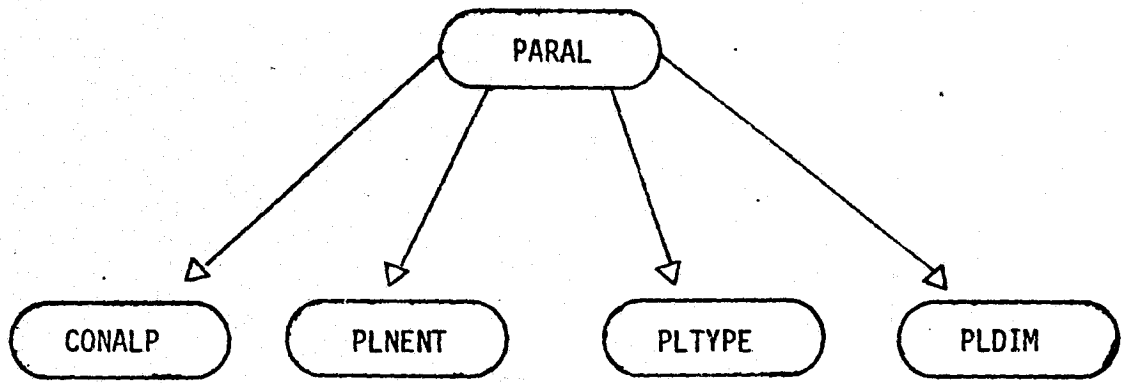
Operations: All information necessary for Parameter List Alignment Check is placed in Alignment Tables. Model for Alignment Check is formal (dummy) parameter list; it is placed in Alignment Table One. Array ARRAY has ARRSIZ entries; each entry specifies which Paramete List Alignment Check is to be performed and whether flags may be produced for primary or secondary listings.

Algorithm: See Source Code Listing.

Parameters: ARRAY - Input

ARRSIZ - Input

See also /ALT/ and "Alignment Tables".



Parameter List Alignment Check,

excluding General Purpose Utilities

SUBROUTINE PARSER

Mnemonic Origin: Parser Routine.

Classification: FFE Parsing Control Routine.

Purpose: Control the parsing of a single Module of FORTRAN code producing tables for later analysis.

Operation: Upon entry, another module of FORTRAN is available for analysis. The module must be processed and analysis tables and source code catalogued for the evaluation and report.

A module number is assigned for the module to allocate table file space on the analysis Table File. If maintenance printing is active, the printer is page restored for the start of the module.

The system is reset in a cyclic fashion until a proper module header is discovered. If stray source code is found between modules, this action will flush the source text. The appearance of a valid header card is detected by a statement process routine assigning a module name to the current module description. For protection, the failure of the statement number to advance will cause loop exit; this will occur if the source code is exhausted while looking for the module header card.

If a module header card is found, the source code origin of the module on the Source Code Catalogue is recorded and processing of module text begins. PARSER looks for a module with the characteristics of a Module header card followed by a series of FORTRAN statements terminated by an END card or the premature appearance of another module header card (i.e., the current module missing an END card).

When the module processing halts cyclic operation by an END or premature header card, graphical nodes are produced for the tables

generated and the relative card counts are adjusted for the next module.

The next module begins on the Current Card (CURCD) if a normal END card was processed. If a premature header card is detected, the Begin Card (BGNCD) is the first card of the next module. The relative card counts are adjusted such that the first card of the next module will be relative card number 1. Note that the contingency is permitted for comment cards to appear between decks.

The source code origin for the next card is computed using the number of relative cards contained in the current module just processed. The count cards in the module are set in the current module description prior to return to the calling routine.

Upon exit, the source code origin and relative card counts are adjusted for the next module and the local tables for the current module have been completely constructed.

Special Note: Note that a premature header card will be left for processing on the next call to PARSER. Thus, processing routines subordinate to PARSER should allow for this contingency.

SUBROUTINE PATHS(BEGINP, ENDP, FOLBAK, COND)

Mnemonic Origin: Follow Paths

Classification: AIR Special Purpose Utility
(Referenced only during search for local uninitialized variables and for DO loop index variables used after loop terminated normally.)

Purpose: Follow non-circular intra-modular paths.

Operation: Build non-circular path from beginning point BEGINP to ending point ENDP, one point at a time. FOLBAK indicates whether PATHS is following or backtracking along path. COND indicates condition under which control was returned to calling routine.

Algorithm: See Source Code Listing.

Condition Codes: See Source Code Listing.

Parameters: BEGINP - Input
ENDP - Input
FOLBAK - Input/Output
COND - Input/Output

Designer's Comment: Double Stack technique used in PATHS should be replaced by single stack technique of TRACHI.

See also "Flow of Control Path Tracing" and Path Stack.

SUBROUTINE PLDIM (PARAM1, PARAM2)

Mneumonic Origin: Parameter List Dimensionality Mismatch

Classification: AIR Query

Purpose: Searches for correspodning parameters not having compatible dimensions.

Operations: If PARAM1 equals 520, warning flags may be produced for primary listing. If PARAM2 equals 521, warning flags may be produced for secondary listing.

Incompatible dimensions are defined as follows:

1. Actual parameter is array and dummy (formal) parameter is scalar.
2. Actual parameter is element of array and dummy parameter is entire array, except
   a) when both parameter have same number of parameters and subscripts of actual parameter are all ones.
3. Parameters do not have same number of dimensions.
4. Parameters have same number of dimensions, but dimensions are not idnetical, except
   a) when dummy parameter has "dummy" dimensions
   b) when dimensions of dummy parameter are all ones

Scalers are defined to have zero dimensions.

Algorithm: See Source Code Listing.

Parameters: PARAM1 - Input
PARAM2 - Input

See also /ALT/ and "Alignment Tables"

SUBROUTINE PLNENT (PARAM1, PARAM2)

Menumonic Origin: Parameter List Number of Entries Mismatch

Classification: AIR Query

Purpose: Searches for corresponding parameter lists not having same number of parameters.

Operations: If PARAM1 equals 500, warning flags may be produced for primary listing. If PARAM2 equals 501, warning flags may be produced for secondary listing.

Algorithm: See Source Code Listing

Parameters: PARAM1 - Input

PARAM2 - Input

See also /ALI/ and "Alignment Tables".

SUBROUTINE PLTYPE (PARAM1, PARAM2)

Mneumonic Origin: Parameter List Type Mismatch

Classification: AIR Query

Purpose: Searches for corresponding parameters not having identical types.

Operations: If PARAM1 equals 510, warning flags may be produced for primary listing. If PARAM2 equals 511, warning flags may be produced for secondary listing.

Algorithm: See Source Code Listing

Parameters: PARAM1 - Input

PARAM2 - Input

See also /ALI/ and "Alignment Tables".

SUBROUTINE POP(DUMMY)

Mnemonic Origin: Pop Control Stack

Classification: AIR General Purpose Utility

Purpose: Delete information at top of Control Stack.

Operation: Delete information at top of Control Stack. If table just deleted does not appear Control Stack, set current row pointer to table to zero. If table just deleted appears elsewhere in Control Stack, set current row pointer to table to value of pointer column of table's occurrence nearest top of Control Stack. Attempts to pop empty Control Stack halts AIR processing.

Parameters: DUMMY - Unused

See also PUSH and Control Stack.

SUBROUTINE PPTRIP

Mnemonic Origin: Postprocess TRIP Table.

Classification: FFE Table Generation Routine.

Purpose: Convert references to statement labels in the transition pairs table to the statement number where the label was defined.

Operation: Upon entry, the transition pairs table (TRIP) contains entries composed of node numbers (statement numbers), special codes, and references to statement labels. Entries containing references to statement labels are to be converted to the node number (statement number) where the label is defined.

Entries containing statement label references are discovered by examining the postprocessing code appended as a flag to the predecessor entry of the transition table entry. The postprocessing flag, predecessor specification, and successor specification are extracted from each entry. If the flag indicates a label reference, the pointer to the Symbol Table is set to select the required label. Uses of the label are searched to detect the defining statement number. The appropriate entry is then replaced with the label definition statement number and the postprocessing flag is removed from the entry in the process.

If a label definition cannot be found, an error message is issued for the module indicating the missing label will affect the graphical examination. The missing label entry is replaced with an undefined label code.

SUBROUTINE PREGEN

Mnemonic Origin: Predecessor Generation.

Classification: FFE Table Generation Routine.

Purpose: Upon entry, the transition pairs table has been converted to special codes and node numbers indicating program flow transitions in the module processed. (See PPTRIP.) The transition pairs table is sorted on Successors entries. This forces the grouping of all predecessors to a particular node to adjacent positions in TRIP and all special code successors (i.e., those not attached to a node) to the bottom of the TRIP table.

In processing predecessors to a particular node, the pointer to the last entry of the Predecessor Table is used to record the predecessors prior to insertion. The current pointer to the Predecessor Table is used for insertion of predecessors to a particular node. After all predecessors have been inserted (possibly empty set), the pointer to the last entry is compared to the current pointer to the Predecessor Table. If entries were actually made, the difference indicates the count of predecessors added.

If node predecessors were inserted, the pointer to the first entry and count are inserted in the Node Table position. If no predecessors were found, zero values are placed in the Node Table entries for both the pointer and count values.

Determining node predecessors requires processing both the explicit predecessors (i.e., branch targets) and the implied predecessors (i.e., fall through transitions). The first node is processed to include only

explicit transitions since there is no "previous node". On other nodes, the implied predecessor (if any) is inserted followed by the explicit predecessors (if any). A previous node is an implied predecessor unless it is an explicit branch or termination of processing statement.

If the Predecessor Table is exhausted before all nodes are processed, an error process is executed. The error process distinguishes overflow cases where the overflow occurred on the last node from overflow on nodes before the last node. In the second case, the remaining nodes are set to indicate no predecessors are recorded.

SUBROUTINE PRIMR

Mnemonic Origin: Primary Report

Classification: Report Generator report processor

Purpose: Control the production of primary reports.

Operation: Upon entry, Primary Reports for flagged modules are to be produced. The Primary Report set will be empty if the Flag option is selected and no flags are present for Primary Reports.

The flag key value for Primary Report flags is set. The current module description is set empty.

Primary reports are produced by a loop, selecting modules for the report then producing the report. The loop is terminated when no more modules are selected.

Module selection is performed by a service routine, reporting by parameter when the report set is complete. If a module is selected, the messages are positioned to the selected module and the module report produced.

After issuing all Primary Reports, the messages are flushed for any remaining flags which were not processed. This action is protection code to prevent malfunction of the Primary Report process from influencing Secondary Reports.

SUBROUTINE PRSKWD

Mnemonic Origin: Parse Key Word statements

Classification: FFE parsing control routine

Purpose: Direct the processing of FORTRAN statements which are identified by a Keyword (i.e. do not contain a zero level equal sign).

Operation: Upon entry, the Parsing Tables are positioned to the first word of a FORTRAN statement which does not contain a zero level equal sign (i.e. an equal sign outside parenthesis). The first entry of the statement should be a FORTRAN Keyword.

The first four characters of the leading word are presented for classification as a FORTRAN Keyword. If the word is not recognized as a Keyword, the current statement type is marked as unrecognizable and a message issued. The Parsing Tables are positioned to the end of the statement to terminate processing of the unrecognized form.

If a valid Keyword is identified, the appropriate parsing routine for processing the statement is selected. Simple statement forms are processed directly by PRSKWD; more complex forms require calling a statement processor for the appropriate type. The return index for the Keyword is used to establish the statement type code of the current statement. This assignment will be modified only by a FUNCTION declaration with a leading type specification. In this case, PRSKWD will initially assign a type

statement code which is reversed to a FUNCTION type later.

If the statement is a header card (e.g. SUBROUTINE, PROGRAM, BLOCK DATA) processing is halted unless this is the first statement of the module. This mechanism detects the premature appearance of another module header. In this event, the statement type code is maintained, but the statement is not processed. Rather, control is returned with the Parsing Tables positioned to the first entry for later processing.

FUNCTION header cards require special treatment. Since the FUNCTION may be preceded by a type declaration, entry to the FUNCTION statement processor may occur either through PRSKWD or indirectly through preliminary analysis of the type declaration. For this reason, the FUNCTION statement processor is required to check for the premature header condition. In addition, the linkage through type statements entry, requires the function name to the extracted prior to calling FUNCI for interface compatibility. Processing FUNCTION declarations is the only process which overrides the statement type assigned by PRSKWD.

SUBROUTINE PRSSTM

Mnemonic Origin: Parse Statement

Classification: FFE parsing control routine

Purpose: Process a single FORTRAN statement

Operation: Upon entry, the Parsing Tables may be empty or occupied. If they are occupied, the Parsing Tables may contain the latter part of a two part statement (e.g. IF( ) A = B) or a statement not processed on the last call (e.g. a premature module header).

The first Use Table position is anticipated for the statement and a new statement number assigned for the statement to be processed. The statement type of the current statement is set to zero to detect statements not being processed.

If the Parsing Tables are empty or exhausted, new statement text is requested. If the new statement is labeled, the label definition is recorded.

If an empty statement is detected from the scan process, the end of input source code has occurred and no statement text is available for processing. Otherwise, normal processing is accomplished on the statement.

If maintenance printing is active, the Parsing Tables are displayed after processing labels.

The first card of the statement is recorded in the current statement description and the statement text is processed. Statement processing is governed by whether the text contains a zero level equal sign (i.e. a equal sign not enclosed in parenthesis).

After processing the statement text, the uses recorded are inspected. If no uses were recorded, the first use indicator is set empty. If a statement was produced, the last card number relative to the module is inserted in the current statement position and a Node Table entry is made to record the statement type, Use table pointer, and card counts. (Note: Node predecessors and successors are not yet determined. These elements are created after the module is completed.)

Control returns with the statement parsing tables left in the last position remaining from statement processing routines.

SUBROUTINE PRSZEQ

Mnemonic Origin: Parse Zero Level Equal Sign statements

Classification: FFE parsing control routine

Purpose: Control processing of FORTRAN statements containing a zero level equal sign.

Operation: Upon entry, a FORTRAN statement occupies the Parsing Tables which contain a zero level equal sign (i.e. a equal sign not enclosed in parenthesis). The statement might be a simple assignment statement, an assignment statement appended to a logical IF statement, a DO statement, or a statement function definition. PRSZEQ determines the general type of statement and passes control to the appropriate statement processing routine.

The first decision is made on whether the form is "variable name(". If this form is found, the statement is not a DO statement or an assignment to an unsubscripted variable. If the variable name is IF, the statement form is either an IF statement, statement function definition with function named IF, or assignment to an array named IF. These cases are distinguished by searching for the form "IF()=". If this form is found, the statement is an assignment statement or statement function; otherwise, it is an IF statement. (Note: this search is only required for IF statement forms which contain assignment statements for conditional execution.)

If the variable name in the form "Variable(" is not IF, the statement is either an assignment to an array element or a statement

function definition. Assignments are distinguished from statement functions by looking for a declared array with the specified variable name. If a declared array is not found, the statement is processed as a statement function definition. If an array name is found the statement is processed as an assignment statement.

For statements where the initial form is not "variable(" the statement is either a DO statement or assignment statement. Since the characters DO may be the leading symbols of a variable name, the cases are distinguished by looking for the index control variables following the equal sign. If the form "DO .... = name," or "DO .... = +name," is found, the statement is processed as a DO statement. Otherwise, an assignment statement is processed.

Before control is passed to the statement processor, the statement type is assigned for the current statement based on analysis of statement type.

SUBROUTINE PRTDIR

Mnemonic Origin: Print Directory

Classification: Control Driver User display routine

Purpose: Display contents of Directory for User information

Operation: The contents and status of Directory entries are displayed for the user's information. The display includes the status of the Directory (i.e. amount of directory space filled and remaining), the module names entered in the Directory, and the description of the modules recorded. If the Directory is empty, the user is informed that no modules are currently recorded. Otherwise, the contents of the Directory are extracted and displayed.

Description of each module includes the name of the module and module characteristics. Modules characteristics include whether the module is a Function Subroutine, or main program; if a subprogram is a secondary entry, this information is conveyed in the module type displayed. The description is obtained by interpreting the module type code through a read-only table of alphabetic symbols set in Block Data.

If the module is defined, a nonzero count of source code cards is displayed.

SUBROUTINE PRTEXP

Mnemonic Origin: Print Explaination

Classification: Report Gernerator Primary Report service routine

Purpose: Print an explaination of flags for a module Primary Report.

Operation: Upon entry, a set of inline user messages (possibly empty) have been produced for a Primary Report. Some flags require additional information explaining their origin or potential effect.

Flag occurance has been marked in the explaination vector. The contents of the explaination vector are searched for a non-empty entry. If at least one entry is found in the vector, a series of explainations are provided for flags which require additional support text. Not all flags require this activity.

SUBROUTINE PRTMSG

Mnemonic Origin: Print Message

Classification: Report Generator service routine

Purpose: Compose a text message to inform the user of a flagged condition.

Operation: Upon entry, a listing report message is available for display to the user. The current message contains flag identification and supporting data for display. The message is interpreted based upon the flag number of the message. Each flag number constitutes an independent case for processing.

The case for processing is selected using a ladder search using the flag number. When the proper flag number is recognized, a write statement is executed to produce the message. After processing the message, the length of the current message is set empty to inform the message construction routine that the data was consumed.

If the ladder search reaches the end of the list without match, an invalid flag number is present. A warning is printed and the message is discarded by setting the length empty.

Special Note: Processing some messages may require not printing descriptive lines. These messages only force lines of code to be displayed. The descriptive write is not required.

SUBROUTINE PRTPRM

Mnemonic Origin: Print Primary Report

Classification: Report Generator Primary Report processor

Purpose: Produces the Primary Report for a single module.

Operation: Upon entry, a module has been selected for which a Primary Report is required. The module may or may not have flagged lines of source code.

The printer page is restored and the header printed. The flag explaination vector is cleared and the line number for source code set empty. The explaination pointer is set empty to detect the appearance of flags after the report is produced.

To produce the report, a loop is implemented. If no flags are present for the selected module, loop iterations are not performed. If flags are present, source lines are printed from the current position to the end of the flagged lines and the current position is advanced. The flag message is printed and the appropriated flag occurance is recorded in the explaination vector. The next message is acquired and the loop repeats. When a message is detected which does not belong to the current module, the looping process is terminated.

To complete the source code listing, source lines are printed from the current position to the end of the module. After printing the listing, the explaination of flags is produced

for the module. This action is taken only if flags were encountered in the source processing.

If there are no flags for the selected module, the looping process is not executed; rather control passes directly to the terminal procedure. This process prints the entire source code body for the module. This action is taken for unflagged modules listed resulting from the ALL option on the REPORT command.

Special Notes: Notice that more than one message may be needed for the same source code line set. By marking the current line, later messages will not produce repeated line prints. The line processing routine will recognize the described set as an empty set.

SUBROUTINE PRTQLS

Mnemonic Origin: Print Query List

Classification: Control Driver user Print routine

Purpose: Inform the user of Queries constructed from Query specifications.

Operation: Contents of the AIR List Table are displayed for user inspection. The List Table contains selected queries corresponding to actions about to be taken and empty (i.e., zero valued) entries for queries turned off by the user.

SUBROUTINE PUSH

Mnemonic Origin: Push Control Stack (opposite of POP)

Classification: AIR General Purpose Utility

Purpose: Add information to top of Control Stack.

Operation: Add information to top of Control Stack from Module Number Register (MR), Table Name Register (TR), List Indicator Register (LR), and Pointer Register (PR).

If table being referenced is global table, module number stored at top of Control Stack is zero.

See also PUSH, Control Stack, and /SPEREG/

SUBROUTINE RDFLAG

Mnemonic Origin: Read Flag

Classification: Physical I/O routine

Purpose Place the next flag information from the Flag File in the flag buffer.

Operation: Upon entry, the next flag is required from the Flag File. Physical reading is dependent upon the status of the Flag File. If an end of file previously occurred, no information is read; rather an empty flag is returned in the I/O buffer to inform the calling routine that no more flags are available.

If end of file has not occurred, Flag data is read from the file, and the buffer is set full. If an end of file occurs on this read, the buffer is reset empty and the control variable set to prevent future reads. This indicator must be reset externally before physical I/O will be permitted again.

SUBROUTINE RDSCAT

Mnemonic Origin: Read Source Code Catalogue

Classification: System I/O service routine

Purpose: Read a card image from the Source Code Catalogue.

Operation: The next record from the Source Code Catalogue is read into the source code buffer. The next record has been established by the pointer in the SCAT file COMMON Block. The pointer is advance to the next record by the read process.

If the end of the source code file has been reached, the end of file indicator is set. The file is protected from over-reading past the file end.

SUBROUTINE RECDIR(NAME, MTYP, MNUM, ORIG, SRCFND)

Mnemonic Origin: Record Directory entry

Classification: FFE table generating routine

Purpose: Insert a module description in the Directory.

Operation: Upon entry, a module name is presented for recording in the Directory. The module description may be a module definition or a module reference to either previously defined or currently undefined module.

Processing the presented name requires three decisions:

1. Is the name a new entry in the Directory or is space already allocated for the name? This decision is based upon a search of Directory entries.

2. Should the current description be inserted in the Directory? Insertion is required if a New entry is presented or if the presented name is a module definition (i.e. not a simple reference to a module). Definitions are distinguished as module names assigned a positive integer module number (i.e. there are analysis tables for the module).

3. Is the current entry a reference or definition entry? The current entry is a reference entry is the module number is zero or there is not current entry under the module name. Note that the Directory pointer is set to the proper insertion point by the searching process.

If the module is new, space must be created in the Directory to accommodate the new description. Otherwise, the current entry will be used to insert the module description. Preparing space for new modules requires moving modules down in the Directory from the insertion point. If the insertion point is below the last active entry, actual movement is not required.

After preparing Directory space (if necessary), the new description of the module is inserted. If the entry is currently occupied by a defined module, the insertion is a replacement of a previous definition. This action is reported for user information.

Parameters:

NAME - module name passed in 2A4 format.

MTYP - module type code

MNUM - assigned module number assigned for access to Analysis Table File data. Presented as value 0 if reference rather than definition.

ORIG - Source code origin for source code card images on Source Code Catalogue. Presented as value0 if only a reference to a module name.

SRCEND - Number of card images recorded for module source code if definition of module; passed as value 0 on simple references.

SUBROUTINE RECSYM(NAME, LENGTH, TYPE, CLASS)

Mnemonic Origin: Record Symbol

Classification: FFE Table generation routine

Purpose: Position the Symbol table to the indicated symbol and record new symbol in the Symbol Table

Operation: Upon entry, a symbol is presented which has been encountered in the processing of a FORTRAN statement. The symbol may be a new symbol or a reference to an existing symbol. If this symbol is new, the symbol is recorded in the Symbol Table; the Symbol Table is positioned to the insertion point in the process. If the symbol is already recorded, the Symbol Table is positioned to the recorded position in the searching process.

The Symbol Table is searched for a symbol compatible with the provided description. If a compatible symbol is found in the search, the Symbol Table is left positioned to the matching table location,

If a compatible symbol is not detected by the search process, the symbol will either be inserted or found to be compatible with a similar existing symbol. The presented symbol may not match an existing entry due to an ambiguity in the process. For example, the symbol may presently be recorded as a scalar variable but used as a function reference; the difference between a scalar variable and function class is an ambiguity which needs resolution.

If a symbol is an ambiguous reference, the character string is the same as an existing symbol. The Symbol Table is search for an entry which has the same character string (i.e. symbolic name) ignoring type and class. If this condition is detected, the ambiguous symbol process is initiated; ambiguity resolution will either detect an ambiguous case or result in symbol insertion.

If the symbol string is not already in the Symbol Table, a new entry is made. The class code is assumed to be resolved by insertion time, but the type code may nct yet be assigned (i.e. for example, in the case of variable names). If the type code is not yet assigned, type is established using the assigned class code and characteristics of the symbolic name. The developed description is then placed in the Symbol Table. Note that the typing operation does not modify the passed type specification; this prevents modification of the presented parameter.

Parameters: NAME - input parameter containing the symbolic name in A4 format.

LENGTH - Length of the symbolic name expressed as the number of integer words required to hold the character string.

TYPE - type specification indicating a type code (positive, nonzero value) or defaulted type (zero value).

CLASS - class code for the required symbol.

SUBROUTINE RECUSE (USECOD)

Mnemonic Origin: Record Use

Classification: FFE Table generating routine

Purpose: Record the presented Use code in the next available Use Table position and link uses among each other and back to the Symbol Table.

Operation: Upon entry, a Use is presented for the symbol currently selected in the Symbol Table. The Use is to be recorded in the Use Table and the position linked to previous Uses of the symbol; linkage points back to the Symbol Table for the first Use.

If Use Table space is available, a Use Table position is allocated for recording the Use. Otherwise, an overflow condition is reported and the Use is discarded.

If Use Table space is available, the Use is placed in the Use Table and appropriate linkage is generated based upon the Use code. Use linkage is dictated by whether the Use is a bracket code or a normal Use code. If an invalid Use code is detected, an anomoly is reported and the Use is recorded with normal linkage.

Parameters: USECOD - input parameter indicating the Use code to record for the currently selected symbol.

SUBROUTINE RDFLGF (FWORD, FLGSIZ, FLAG, UNFLAG)

Mnemonic Origin: Read Flagged Full Word

Classification: FFE bit manipulation routine

Purpose: Separate the data and flag components of a flagged full word integer.

Operation: Upon entry, a flagged full word integer is presented along with a description of the flag size. The flag bits are located in the MSB positions of the word; the LSB of the word contain other data. RDFLGF separates the flag from the data and returns the separated results right justified with zero left fill.

RDFLGF is a machine dependent routine. The flag value is extracted by clearing data bits from a copy of the presented word, moving the flag value to the right most positions. Left bit positions are set to zero.

The data field is extracted by clearing the flag field in a data word copy with zeroes, clearing the contribution of the flag.

Parameters: FWORD - Full word integer composed of flag bits and data bits in adjacent fields.

FLGSIZ - integer input parameter describing the number of MSB positions which constitute the flag bits.

UNFLAG - output return parameter into which the data bits are places with flag bits positions set to zero.

SUBROUTINE RDFLGH (HWORD, FLGSIZ, FLAG, UNFLAG)

Mnemonic Origin: Read Flagged Half Word

Classification: FFE bit manipulation routine

Purpose: Separate the flag bits from the data bits of a half word data value.

Operation: Upon entry, a data value is presented which contains information bits in the lower half word positions. The information is composed of a flag in the MSB position of the lower half word followed by data bits in the LSB bits.

RDFLGH is a machine dependent routine. The flag bit values are separated from the data bits by clearing the upper half word then moving MSB field to the right clearing the left bits of the results to zero left fill.

The data bits are extracted from the lower half word in LSB positions of the half word clearing the bits containing flag information to zero values.

Parameters: HWORD - integer input parameter contain information in the lower half word composed of a flag field followed by a data field. Upper half word contents are unknown.

FLGSIZ - description of flag field size expressed as the number of bits in the lower half word containing flag information.

FLAG - output parameter containing the flag value extracted from the half word returned right justified with zero left fill.

UNFLAG - output parameter containing the unflagged data present in the lower half word of flagged information.

SUBROUTINE READLT(MODNO)

Mnemonic Origin: Read Local Tables

Classification: AIR General Purpose Utility

Purpose: Bring local tables of module into main memory.

Operation: Bring local tables for module with module number MODNO into main memory from secondary storage. If module does not exist, zero out local tables in main memory.

Parameters: MODNO - Input

SUBROUTINE REDCOL (CLASS)

Mnemonic Origin; Reduce Comma List

Classification: FFE parsing support routine

Purpose: Reduce the complexity of actual parameters and subscripts to single operands.

Operation: Upon entry, the Parsing Tables are positioned to the name of a FORTRAN structure having a comma separated access list enclosed in parenthesis. The FORTRAN structure may be an array reference, subroutine call, function reference, or statement function reference.

If the access list consists of simple operand entries, no action is taken; the list is simply examined. If the access list contains arithmetic expressions, functions references, or nested array references used as subscripts, the access list is reduced by processing the member and replacing it with a simple operand. If the replaced member is a function reference , the function name is substituted for the call. If the member is an arithmetic expression, the expression is processed and replaced with a temporary name. For example, the form,

CALL SUB (1, A+2, FUNC(3,4))

would be reduced to,

CALL SUB (1, temp, FUNC)

Complexity of REDCOL results from the need to process recursive forms in a nonrecursive language. A stack is maintained

to facilitate recursion on functions in functions, arrays within arrays, etc. The stack records the processing state as elements are processed. If a new structure is encountered, a recursive call is effected to this routine again. In the call, the stack is pushed down to begin a new processing description. The stack is pulled when the final ")" is encountered. If no recursive forms have been processed, a normal return is performed to the calling routine. If a recursive form has been found, the process returns to the next outer form and processing resumes on the form.

Reduction stack entries contain the following information:

1. Class code of the structure
2. Parsing Table position of the structure name
3. Parsing Table position of the first element of the current member being processed. Advanced as members are processed in the comma list.
4. Need for subexpression indicator. Set if a member is found to contain arithmetic expression.
5. Relative parenthesis count. Used to distinguish parenthesis of arithmetic expression from final right parenthesis of the structure.

Items 1 and 2 are maintained during the processing of a structure. Items 3 through 5 are modified as the processing proceeds from one member of the comma list to the next.

To reduce the access list, individual members are examined

one at a time. The Parsing Table position of the first element of the member is recorded in the stack for possible later processing.

Elements of the member are examined one at a time. If an arithmetic expression symbol is encountered, the stack subexpression indicator entry is set to indicate that at the end of the member, a subexpression should be processed. When the next comma or final right parenthesis is encountered, the member has been spanned. If an arithmetic expression was encountered in the review process, the expression is processed as a subexpression and the expression replaced by a temporary name.

Once all members have been processed, the class of the name symbol is examined. If the name is a function or statement function reference, a function reference is processed. The parameter list of the function is removed, leaving only the function name in the access position.

Before returning control, the Parsing Tables are repositioned to the name entry of the first structure processed. Upon return, the structure has been simplified so the calling routine can continue processing the structure it is working on.

Example. To illustrate the comma list reduction process. an example of various states in the process is shown. The Parsing Tables and reduction stack are shown as the structure is processed. Suppose the statement is an assignment statement of the form,

```
R = B( 1, J+K,  FUNC(3,5/R)+3 )
```

The reduction process is applied to the array reference for B in processing the arithmetic expression.

The position of B and the class code of "array" is entered in the stack entry. The position of "1" is recorded as the member location starting position and processing proceeds to the first comma. Since a simple operand is found, no subexpression is required.

Processing moves to the second subscript and the position of "J" is recorded in the stack. Since the "+" indicates an expression, the member is marked for replacement. Upon reaching the second comma, the subexpression "J+K" is processed and replaced with the temporary "I*TM A".

Processing then moves to the next subscript, an expression involving a function call. Upon detecting the "FUNC", a recursive call is effected, opening a new stack entry. The Parsing Table position of "FUNC" and a class of "function" is recorded in the stack. Processing of the actual parameter list then proceeds. In a similiar fashion, the actual parameter "3" is scanned and the expression "5/R" replaced by a temporary.

At the end, the function reference is processed and the Parsing Tables modified to delete the actual parameter list, leaving the function name.

Control then returns by a recursive return popping the stack entry. In this fashion, processing of the subscript list continues. The subexpression "FUNC + 3" is processed and

replaced by the temporary "R*TM C".

Before return control to the calling routine, the Parsing Tables are repositioned to the "B" entry and the form returned is,

R = B(1, I*TM A, R*TM C)

Parameters:

CLASS - class code of the outermost structure presented for reduction.

Special Notes: REDCOL is given the capability to process function references used in access lists. This capability is also exploited for processing function references passed for reduction on the outermost level. If a function reference "FUNC (A+1, B)" is passed for actual parameter list reduction, the form returned to the calling is simply the function named "FUNC".

Note that subscripted array references which appear as subscripts are reduced to subexpressions even though no arithmetic operations are applied to the array element. If the array name is presented without subscripts, no action is taken.

Warning. REDCOL is used by the arithmetic expression processing routine. Support processing for arithmetic expressions discovered by REDCOL are limited to "simple arithmetic expressions" to prevent a cyclic calling pattern.

```
P = B( 1, J+K, FUNC(3, 5/R) + 3 )
```

INTERMEDIATE SYMBOL STRING
LENGTH = 300
LAST ENTRY = 22
CURRENT POINTER = 1
ZERO EQUAL SIGN = 2

	INDEX	CONTENTS
**	1	V
	2	=
	3	V
	4	(
	5	I
	6	,
	7	V
	8	+
	9	V
	10	,
	11	V
	12	(
	13	I
	14	,
	15	I
	16	/
	17	V
	18	)
	19	+
	20	I
	21	)
	22	$$

TEMPORARY SYMBOL TABLE
LENGTH = 300
LAST ENTRY = 10
CURRENT POINTER = 1

	INDEX	CONTENTS
**	1	P
	2	B
	3	1
	4	J
	5	K
	6	FUNC
	7	3
	8	5
	9	R
	10	3

START OF FORM F

INTERMEDIATE SYMBOL STRING
LENGTH = 400
LAST ENTRY = 22
CURRENT POINTER = 3
ZERO EQUAL SIGN = 2

TEMPORARY SYMBOL TABLE
LENGTH = 300
LAST ENTRY = 10
CURRENT POINTER = 2

	INDEX	CONTENTS
	1	V
	2	=
**	3	V
	4	(
	5	I
	6	,
	7	V
	8	+
	9	V
	10	,
	11	V
	12	(
	13	I
	14	,
	15	I
	16	/
	17	V
	18	)
	19	+
	20	I
	21	)
	22	$$

INDEX	CONTENTS
1	R
** 2	B
3	I
4	J
5	K
6	FUNC
7	3
8	5
9	R
10	3

REDUCTION STACK STATUS LEVEL = 1

CLASS	VAR LOC	PGN MEMB	NEED SBE	REL PRN CNT
3	3	6	0	0

BEGIN PROCESS FOR MEMBER BEGINNING 1

REDUCTION STACK STATUS LEVEL = 1

CLASS	VAR LOC	PGN MEMB	NEED SBE	REL PRN CNT
3	7	6	0	0

BEGIN PROCESS FOR MEMBER BEGINNING J

REDUCTION STACK STATUS LEVEL = 1

CLASS	VAR LOC	PGN MEMB	NEED SBE	REL PRN CNT
3	3	7	0	0

REPLACING MEMBER EXPRESSION BETWEEN 7 AND 9
WITH TEMPORARY I*TM A

INTERMEDIATE SYMBOL STRING
LENGTH = 400
LAST ENTRY = 20
CURRENT POINTER = 7
ZERO EQUAL SIGN = 2

TEMPORARY SYMBOL TABLE
LENGTH = 300
LAST ENTRY = 9
CURRENT POINTER = 4

	INDEX	CONTENTS
	1	V
	2	=
	3	V
	4	(
	5	I
	6	,
**	7	V
	8	,
	9	V
	10	(
	11	I
	12	,
	13	I
	14	/
	15	V
	16	)
	17	+
	18	I
	19	)
	20	$$

INDEX	CONTENTS
1	R
2	B
3	I
** 4	I*TM A
5	FUNC
6	3
7	5
8	R
9	3

```
BEGIN PROCESS FOR MEMBER BEGINNING FUNC

REDUCTION STACK STATUS        LEVEL =      1
    CLASS  VAR LOC  PGN MEMB  NEED SBE  REL PRN CNT
        3        3        9          0          5
      START OF FORM FUNC     (


   INTERMEDIATE SYMBOL STRING                TEMPORARY SYMBOL TABLE
   LENGTH =    400                           LENGTH =    300
   LAST ENTRY =     22                       LAST ENTRY =      9
   CURRENT POINTER =      9                  CURRENT POINTER =      5
   ZERO EQUAL SIGN =      2

   INDEX  CONTENTS                    INDEX  CONTENTS

        1  V                              1  P
        2  =                              2  R
        3  V                              3  1
        4  (                              4  I*TM     A
        5  I
        6  *
        7  V
        8  ,
**      9  V                        **     5  FUNC
       10  (
       11  I                              6  3
       12  ,
       13  I                              7  5
       14  /                              8  R
       15  V
       16  )
       17  +
       18  I                              9  3
       19  )
       20  $

REDUCTION STACK STATUS        LEVEL =      2
    CLASS  VAR LOC  PGN MEMB  NEED SBE  REL PRN CNT
        3        3        5          0          0
        4        9        8          R          0
  BEGIN PROCESS FOR MEMBER BEGINNING 3

REDUCTION STACK STATUS        LEVEL =      2
    CLASS  VAR LOC  PGN MEMB  NEED SBE  REL PRN CNT
        3        7        9          0          0
        4        9       11          0          0
  BEGIN PROCESS FOR MEMBER BEGINNING 5

REDUCTION STACK STATUS        LEVEL =      2
    CLASS  VAR LOC  PGN MEMB  NEED SBE  REL PRN CNT
        3        7        9          0          0
        4        9       13          0          0


  REPLACING MEMBER EXPRESSION BETWEEN      13 AND      15
    WITH TEMPORARY R*TM   8
```

REPLACING FUNCTION WITH FUNCTION NAME

INTERMEDIATE SYMBOL STRING
LENGTH = 400
LAST ENTRY = 18
CURRENT POINTER = 15
ZERO EQUAL SIGN = 2

INDEX	CONTENTS
1	V
2	=
3	V
4	(
5	I
6	,
7	V
8	,
9	V
10	(
11	I
12	,
13	V
14	)
15	*
16	I
17	)
18	↑

RECURSIVE RETURN

TEMPORARY SYMBOL TABLE
LENGTH = 300
LAST ENTRY = 8
CURRENT POINTER = 8

INDEX	CONTENTS	
1	D	
2	3	
3	1	
4	I*TM	A
5	FUNC	
6	2	
7	2*TM	2
8	3	

```
REDUCTION STACK STATUS        LEVEL =      1
    CLASS  VAR LOC  PGM MEMB  NEED SRE  REL PRN CNT
        3        3         9          0            0

  INTERMEDIATE SYMBOL STRING              TEMPORARY SYMBOL TABLE
  LENGTH =    400                         LENGTH =    300
  LAST ENTRY =     12                     LAST ENTRY =      6
  CURRENT POINTER =     10               CURRENT POINTER =      6
  ZERO EQUAL SIGN =      2

  INDEX  CONTENTS                 INDEX  CONTENTS

      1  V                            1  P
      2  =                            2  P
      3  V                            3  1
      4  (                            4  I*TM     A
      5  I                            5  FUNC
      6  ,                            6  3
      7  V
      8  ,
      9  V
**   10  +                    **
     11  I
     12  )
     13  $$
 REPLACING I*TM  EXPRESSION BETWEEN      9 AND     11
  WITH TEMPORARY P*TM    C

  INTERMEDIATE SYMBOL STRING              TEMPORARY SYMBOL TABLE
  LENGTH =    400                         LENGTH =    300
  LAST ENTRY =     11                     LAST ENTRY =      5
  CURRENT POINTER =      9               CURRENT POINTER =      5
  ZERO EQUAL SIGN =      2

  INDEX  CONTENTS                 INDEX  CONTENTS

      1  V                            1  P
      2  =                            2  P
      3  V                            3  1
      4  (                            4  I*TM     A
      5  I                            5  P*TM     C
      6  ,
      7  V
      8  ,
**    9  V                    **
     10  )
     11  $$
```

PARSING TABLES AFTER REDUCTION PROCESS

INTERMEDIATE SYMBOL STRING
LENGTH = 400
LAST ENTRY = 11
CURRENT POINTER = 3
ZERO EQUAL SIGN = 2

	INDEX	CONTENTS
	1	V
	2	=
**	3	V
	4	(
	5	T
	6	,
	7	V
	8	,
	9	V
	10	)
	11	$$

TEMPORARY SYMBOL TABLE
LENGTH = 200
LAST ENTRY = 5
CURRENT POINTER = 2

	INDEX	CONTENTS	
	1	2	
**	2	3	
	3	1	
	4	INTV	A
	5	INTV	C

SUBROUTINE REDLOP

Mneumonic Origin: Redifinition of DO Loop Control Variables

Classification: AIR Query

Purpose: Searches for DO Loop control variables assigned values within loop itself.

Operations: Program boundaries are not crossed. It is assumed that external references within the DO Loop do not modify control variables.

Warning flags may be produced for primary listing.

Algorithm: See Source Code Listing.

SUBROUTINE RESWRD

Mneumonic Origin: FORTRAN Reserved Words

Classification: AIR Query

Purpose: Searches for FORTRAN "reserved words" being used as names.

Operations: Warning flags may be produced for primary listing.

Algorithm: See Source Code Listing.

SUBROUTINE RPTGEN

Mnemonic Origin: Report Generator

Classification: Report Generation primary control routine

Purpose: Control the production of Reports.

Operation: Upon entry, a REPORT command has been processed and report options selected. Primary Reports are produced followed by Secondary Reports.

The Report Generation process is initialized and the first message is acquired. Primary Reports are then produced. Page restoration separates Primary and Secondary Reports.

Secondary Reports are then produced until an empty message is detected. Secondary Reports are of two types: 1) Secondary Listing reports and 2) Secondary Display reports. These types are distinguished by examining the source code card indicators of the first report message. If source code cards are not required, a Display report is produced. If source is required, the Listing report is produced.

SUBROUTINE SABORT

Mnemonic Origin: Statement Abortion

Classification: FFE statement parsing routine

Purpose: Abort the statement in progress.

Operation: Upon entry, the statement in progress is found to contain an error. Continued processing of the statement might produce invalid and misleading processing or result in malfunction potential for the system. The statement must be aborted to continue processing.

The Parsing Tables are positioned to the construction being processed when the malfunction is detected. The user is informed via a generated error flag that statement processing was halted. The Parsing Tables are positioned to the end of the statement to neutralize further processing.

Since an imbalance in the statement text may be causing the abortion, the Subexpression and Begin/End List stacks are reset to isolate the error to the current statement.

Special Notes: The abortion of a statement does not necessarily cause immediate halt of statement activity. Processing routines may continue examinations on the Parsing Table contents to complete their activities. The abortion process simply conceals any further text consideration by analysis routines.

Note that if the error occurs in the IF condition portion of a logical IF statement, abortion will neutralize processing of the conditional statement.

SUBROUTINE SBELNK (USECOD)

Mnemonic Origin: Subexpression Bracket Linkage

Classification: FFE Table Generating Routine

Purpose: Establish Use Table links for Begin/End Subexpression Bracket Use Codes.

Operation: Upon entry, a Begin or End Subexpression Use Code is to be recorded. Use Table links are required for this entry in the Use Table.

Begin Subexpression Use Codes are processed separately from End Use Codes; a single routine is implemented to surface the interaction among these codes through the Subexpression Bracket Stack.

When a Begin Subexpression Use code is encountered, the backward pointer of the current Use Table position is set empty to cause an independent list of Use Table entries to begin. The current position of the Use Table is inserted in the top of the Subexpression Bracket Stack to record the last Begin Bracket position.

When an End Subexpression Use code is encountered on a later call, the back pointer of the End Bracket is set to the last Begin Bracket position and the forward pointer is set to zero. The forward link of the last Begin Bracket Use Table position is set to the current Use Table position (i.e. the location of the End Subexpression bracket). The top entry of the Subexpression Bracket is removed from the stack.

Parameters: USECOD - The code of either a Begin or End Sub-expression code.

SUBROUTINE SCAN

Mnemonic Origin: Scan a Fortran Statement

Classification: FFE Scan Process Control Routine

Purpose: Control the scanning of a FORTRAN statement to create Parsing Tables.

Operation: Upon entry, the next statement of FORTRAN text is required for processing. Parsing Table entries are to be constructed by scanning the statement text.

The Parsing Tables are set empty to clear the last statement text. The Scan Buffer is set empty to indicate the next card should be an initial FORTRAN statement card (i.e. not a continuation card).

Scanning the statement requires two operations: 1) a preliminary scan operation and, 2) a scan postprocess operation. In the preliminary scan operation, elementary statement constructions are extracted from the FORTRAN Test. In the postprocess, complex constructions recognized as sequences of elementary constructions are combined and the zero level equal sign (if any) is detected.

If a blank card is detected by the preliminary scan, the card image is not returned; an empty card is returned only if the input source is exhausted. An empty card is a card containing only and End of Statement code without any other parsing text to process.

SUBROUTINE SCNPRO

Mnemonic Origin: Scan Process

Classification: FFE Scan control routine

Purpose: Create preliminary entries in the Parsing Tables for a FORTRAN statement.

Operation: Upon entry, the Parsing Table entries are to be constructed for a single FORTRAN Statement. The statement is composed of an initial FORTRAN card and all continuation cards of the statement.

Statement text is processed in a cyclic fashion until either an end of statement code is detected or the Parsing Tables are filled. The lexical item is reset and the reserve pointer to the Scan Buffer set to the current entry (See Scan Buffer operation description).

Based upon the first character of the next Scan entry, an appropriate subprocess is initiated to create a Parsing Table entry. The Parsing Tables are examined to insure at least one additional position is available in both the Intermediate Symbol String and Temporary Symbol Table. The lexical item is then constructed based upon the first character from the Scan Buffer. If the first character indicates a multiple character lexical item control is passed to the appropriate support processor. If the first character is a special symbol which is not the leading character of a multiple character item, the single character is inserted in the Intermediate Symbol String.

The following multiple character lexical items are created by the preliminary scan process:

1. Alphanumeric strings beginning with an alphabetic character.
2. Hollerith literal strings.
3. Numeric literal strings (both decimal and nondecimal based).
4. Relational and logical operators.
5. Logical constant literals.

Notice that Hollerith constants must be processed by the preliminary scan process since information on blank card columns are lost after this process.

If the process terminated by overflow of a Parsing Table, remaining text of the statement is flushed. This procedure positions the source code to the next statement preventing effects of the overflow from influencing the next statement's process.

Finally, an end of statement code is inserted in the Intermediate Symbol String. The code is forced into the Parsing Tables if a processing error has caused all available entries to be used.

Special Note: Notice that the ISS entries are not fully available for statement text. The logical end of ISS pointer is used to control the number of entries allocated to statement elements. In effect, this reserves space at the end of ISS for the end of statement code and a protective buffer of these codes.

SUBROUTINE SCNPST

Mnemonic Origin: Scan Post process

Classification: FFE Scan process control routine

Purpose: Post process the Parsing Table contents to identify and combine complex constructions which appear as a sequence of elementary entries and identify appearance of zero level equal sign.

Operation: Upon entry, the Parsing Tables contain elementary entries produced by the preliminary scan process. These entries are searched for complex forms appearing as a series of elementary entries in the Parsing Tables. If a complex form is identified, the Parsing Table entry is replaced by a combined construction.

The following complex forms are identified in the post process procedure:

1. Floating point constants
2. Complex constants

The Parsing Tables are examined from the first entry to the last nonempty entry for a series of adjacent elementary forms which constitute a floating point or complex constant. Since the scan process is blind to statement context, care must be taken to avoid recognizing FORMAT statement entries as floating point constants. Also, parameter lists should not be recognized as complex constants.

Scanning rules are developed to avoid these errors. Since valid floating point constants in legal FORTRAN statements are

preceded by special symbols, the appearance of the form "variable.number" is not a floating point constant. If this form is found, elements in the Parsing Tables are skipped until a separator other than a period is found. Similarly, two floating point constants passed by parameter will have the form: "variable(F, F)". The appearance of a variable prior to the open parenthesis indicates the structure is not a complex constant.

Floating point constants are produced first in the tables since floating point constants are substructures in complex constants. Recognition of floating points constants key off the appearance of a period or integer. A service routine performs detailed investigation for legitimate forms of floating point constants.

Complex constants key off the trailing right parenthesis of the form. Entries prior to the right parenthesis are searched for a preceding left parenthesis. If the required structural form is detected, a closer investigation is performed for acceptable construction components.

If floating point or complex constants are detected, the elementary elements are replaced with a single entry.

After processing complex forms, the Parsing Tables are searched for a zero level equal sign. This is an equal sign not enclosed in parenthesis. If a zero level equal sign is found, the Parsing Table position of the equal sign is recorded in the zero level equal sign indicator for the statement. The position

marked is the first equal sign in a left to right scan.

After processing the Parsing Table contents, a protective buffer of end of statement codes is inserted in the Parsing Tables.

SUBROUTINE SECNDR

Mnemonic Origin: Secondary Reports

Classification: Report Generator control routine

Purpose: Produce a secondary report in Listing format.

Operation: Upon entry, a message has been detected for which a secondary listing report is required. This format of report requires extraction of source code to be displayed.

The report key value and source code origin are recorded to enable detection of changing reports and modules. When a message is found which differs from the message which began the process, this report is over.

The page is adjusted and header printed. A loop then produces the report until a change in messages is found. Within the loop, a change in source code origin indicates a new module's source code is participating in the display. Space is provided between source code of different modules to increase readability of the report. Additionally, the line count is reset on new modules.

Source lines are printed from either the next line or the first card indicated by the message. This selection permits multiple messages for the same line to produce only one instance of source printing. After printing the source code lines, the current line indicator is advanced to the last card of the message or the current position.

After printing the appropriate source lines, the message content is printed, and the next message retrieved. The process is repeated while messages continue to appear with the same key value.

Upon exit, a message has been detected with a different key. This message is returned to the calling routine for analysis.

SUBROUTINE SELMOD(FINISH)

Mnemonic Origin: Select Module

Classification: Report Generator service routine

Purpose: Select the next module for Primary Report generation.

Operation: Upon entry, the next module is required for Primary Report production. The module is to be selected by considering control options selected through the command card, current message contents, and Source Code Catalogue order. When all modules have been processed for Primary Report production, the condition is communicated to the calling routine.

The primary selection criteria is determined by whether the ALL option is selected or not. If ALL is selected then modules are to be listed whether they are flagged or not. This is accomplished using the source code origin of the current module to determine the nearest neighbor on the Source Code Catalogue. Notice that the absolute card image of the last card in the current module is used rather than the source code origin. This is required to overcome the initial transient where the module desired has a source code origin of zero. Since negative numbers are to be avoided, the current module description is initialized to zero origin, zero relative card numbers. The next module is thus the module beginning with origin zero, relative card number 1.

To search the Directory, the first entry is arbitrarily established as the current "closest" module to the current module. The initial distance is established as a large number which is greater than the largest number of "garbage" cards which are likely to appear between two modules. If garbage cards are detected between modules, the card images are recorded on the Source Code Catalogue during analysis but no module definition is entered in the Directory. These cards cause "gaps" to appear between the end of one module and beginning of another. If no cards appear between decks, the first card of the next module will be one card ahead of the last card of the previous module.

To detect the next module in an ALL option, the Directory entries are searched for the "closest" module. This search is performed by comparing the distance between the Directory entry Source code Origin and the last card of the current module. If a closer entry is found, the Directory pointer is copied to the selected entry and the new distance is established. Finding a closer entry also causes the completion indicator to be reset, enabling continuation of Primary Reports.

If the FLAG option is selected explicitly or by default, module selection is simplier. The next module is the module specified by the next message. If all Primary Report messages are complete, module selection is over. After establishing the source code origin of the next module, the Directory is searched for an entry with the specified origin. If no module

is found with the indicated origin, a warning message is issued and a phantom module description is established as the current module. The Directory pointer is set empty to disable extraction of module description from the Directory.

After module selection process is complete, module characteristics are extracted from the Directory entry. These data items are placed in the current module description prior to returning control to the calling routine.

Parameters: FINISH - logical decision parameter to communicate the end of Primary Report modules to the calling routine.

Special Notes: Caution must be observed in selecting modules from the Directory. Secondary entry points are recorded with the same source code origin and card indicators as the primary entry point. The primary entry point name should be retrieved rather than a secondary entry point.

Additional care is required in selecting the first module (i.e. with source code origin 0) since modules which are only references are recorded with origin value zero. Reference modules can be identified by either having a zero card count (i.e. no source code) or a zero module number entry (i. e. no table file).

SUBROUTINE SETRES

Mnemonic Origin: Set reserve pointer

Classification: FFE scan support routine

Purpose: Set the reserve pointer to the Scan Buffer.

Operation: The reserve pointer is set to the current Scan Buffer position to establish the deletion bound in the event Scan Buffer compression is required. This routine is coded as a small stand alone procedure to prevent proliferation of the Scan Buffer COMMON Block to other routines.

SUBROUTINE SETSCA(VALUE, SCAL, TAB)

Mnemonic Origin: Set Scalar

Classification: AIR General Purpose Utility

Purpose: Allows scalars associated with permanent AIR data structures to be set.

Operations: Algorithm: Binary tree search through permanent data structure names, followed by binary tree search through scalar identifier names (see "AIR Abbreviations"). The desired scalar then receives a value.

Scalars which may be set:

1. current row pointer to data structure (table or stack)
2. pointer to last non-empty (valid) row in data structure

Parameters: VALUE - Input

SCAL - Input

TAB - Input

Scalar indicated by SCAL associated with data structure TAB is set to value of VALUE.

see also GETSCA

SUBROUTINE SHIFTY (OUTVEC, INVEC, CHRPOS, CNT)

Mnemonic Origin: Shifting routine

Classification: FFE bit manipulation routine

Purpose: Shift character strings packed in A4 format.

Operation: Upon entry a vector of packed characters is presented from which a subset character string is to be extracted. The extracted characters are placed in an output vector in A4 format.

Processing begins by verifying the passed parameter description of the characters to be extracted. If an error is detected, an anomoly is issued and the input parameters values are adjusted to the limits of operation.

Character extraction is performed by cyclic operation which halts if the required character subset is spanned, the input vector is exhausted, or output vector is exhausted. This procedure protects against requests for character strings which exceed the physical bounds of the holding data structures.

The input vector position containing the first character of the desired string is selected and a character number within that position computed.

Characters are first selected in sets of 4, selecting a subset of the character string from the current and next position of the input vector. If the input vector is at the boundary entry, blank characters are substituted for the next vector position.

With the transfer of each set of four characters, the output vector is advanced, the count reduced by four, and the input vector advanced. This procedure may cause the count to go negative.

When the count becomes zero or negative, the major loop is abandoned and the last entry is adjusted if needed. If the count is zero a modulo 4 character string was extracted. If the count is negative, more characters were transferred than desired; the amount of the negative count indicates this character surplus. To correct for excess character transfer, the last output entry is extracted and the surplus characters removed and replaced by blanks.

Finally, if the output vector is not completely filled, blank character strings are inserted in the remaining output vector positions.

Parameters: OUTVEC - output vector receiving the selected character substring in A4 format.

INVEC - input vector containing the desired substring in A4 format.

CHRPOS - character position within the input vector indicating the first character in the desired substring.

CNT - character count indicating the length of of the substring expressed as the number of characters.

Special Notes: SHIFTY was an old routine originally developed for FACES version 1. The parameter list order is the same as the original version. The routine was recoded in a more flexible form pursuant to changes anticipated in future versions of the system. Operation will be generalized to simplify character selection allowing other than three dimensional arrays in the newer application.

SUBROUTINE SIDCPX(AT, CPXC)

Mnemonic Origin: Scan Identification of Complex Constants

Classification: FFE Scan Support Routine

Purpose: Determine if a structure of elementary entries in the Parsing Tables is a complex Constant.

Operation: Upon entry, a Parsing Table position is indicated which contains a left parenthesis of a possible complex constant. This routine examines the structure to determine if the Parsing Table structure is a complex constant. Notice that the indicated position is not the current position of the Parsing Table.

Recognition is coded as a series of conditions required for the entries to contain a complex constant. If the Parsing Tables contain an acceptable structure, the interior of the text is reached and the form is accepted; otherwise, failure of any one of the conditions results in an unaccepted form.

If the symbol preceding the left parenthesis is a variable, or there is no preceding symbol, the structure is not a complex constant. Otherwise, the form is inspected for a pair of Floating point constants which may be real or double precision separated by a comma and enclosed in parenthesis. Either or both the constants may be optionally signed.

If the form is identified as a complex constant, the return parameter is set TRUE; otherwise, the return parameter is set FALSE.

Note that only ISS entries are examined in the search and that the pointers to the Parsing Tables are not modified in the examination.

Parameters: AT - Position of ISS containing the left parenthesis of the candidate complex constant form

CPXC - Return parameter indicating whether the form is identified as a complex constant or not.

SUBROUTINE SIDFPC(AT, FPC)

Mnemonic Origin: Scan identification of Floating Point Constant

Classification: FFE scan support routine

Purpose: Examine a structure in the Parsing Tables to identify the presence of a Floating Point constant construction.

Operation: Upon entry, a Parsing Table construction is presented for examination as a floating point constant. The first element of the construction is indicated by parameter. Parsing Table entries are examined to determine if the entry is a floating point constant.

If the entry is a variable character string or there is no preceding entry, the construction is not a floating point constant. A floating point constant is indicated by a integer/fraction specification (i.e. mantissa) with an optional exponent specification (i.e. characteristic). The integer/fraction specification may omit either the integer or the fraction. If an explicit exponent is specified, the decimal point following the integer specification may be omitted.

Parameters: AT - input parameter indicating ISS position of Parsing Tables containing the first element of a potential Floating Point constant.

FPC - output parameter indicating whether the form is a floating point constant or not.

LOGICAL FUNCTION SMATCH(NAME, LENGTH)

Mnemonic Origin: Symbol Match

Classification: FFE table generation support routine

Purpose: Compare the specified symbolic name to the Symbol Table symbolic name of the current symbol table entry.

Operation: Upon entry, a symbolic name is presented for comparison to the current (i.e. pointer addressed) Symbol Table entry. If the symbolic content is identical, a TRUE result is returned.

The symbol must match in both length and content. Therefore, it is first determined if the symbolic content is located in the main Symbol Table entry or the Symbol Overflow Table. If the presented symbol is a standard size but the Symbol Table entry is oversized, no match is present. If both are standard size, the character strings results in a match condition.

If the presented symbol is oversized, the Symbol Table entry must also be oversized also for a match condition. In addition, the length of the symbol string in the Overflow Table must be equal to the presented symbol before a match is possible. If both are the same length, the character content of the Symbol Overflow entry is inspected; if they are the same, a match condition is present.

Notice that an empty Symbol Table position is allowed in the matching process. If a zero valued symbol (i.e. a symbolic name of two zeroes) is presented, an empty Symbol Table position

will match the presented symbol. If a nonempty symbolic name is compared to an empty Symbol Table position, no match will be found.

Parameters: NAME - Symbolic name to be compared presented as a one dimensional vector.

LENGTH - Length of the symbolic name expressed as the number of words in the vector.

SUBROUTINE SNALPH(INCHR)

Mnemonic Origin: Scan Alphanumeric character strings

Classification: FFE Scan support routine

Purpose: Scan lexical items which are alphanumeric character strings beginning with an alphabetic character.

Operation: This routine may be entered directly from the preliminary scan process or another scan service routine. As a result, the lexical item may already contain characters when the routine is entered.

Storage of character strings is different for alphanumeric items than for other lexical items; alphanumeric character strings are stored in groups of 8 characters or less in sequential positions of the Parsing Tables. Overflow entries are not made for these lexical items.

Therefore, processing begins by examining the lexical item. If 8 characters or more are already present in the lexical item, entries are made in the Parsing Tables until the number of lexical item character is less than 8.

Scan characters are processed until a character other than an alphabetic or numeric symbol is encountered. While alphanumeric characters are added, the length of the lexical item is monitored. If the length reaches 8 characters, a Parsing Table entry is made and the lexical item reset empty.

After a terminal character is encountered, the lexical item is examined for residual characters. If characters are detected, the character string (less than 8 characters) is entered in the Parsing Tables.

Parameters: INCHR - First character passed by calling routine to be added to alphanumeric string.

SUBROUTINE SNCPX

Mnemonic Origin: Scan Complex Constant

Classification: FFE scan service routine

Purpose: Scan complex constant construction for Parsing Table insertion.

Operation: Upon entry, a complex constant form has been detected among the elementary entries of the Parsing Tables. The Parsing Tables are positioned to the left parenthesis of the complex constant form. Characters of the complex constant between and including the parenthesis are extracted from existing Parsing Table entries and inserted in the lexical item in a cumulative fashion.

After extracting the complex constant string, the Parsing Table entries containing the elementary symbols are deleted leaving a space for inserting the constructed complex constant. The entry is replaced with a complex constant ISS code and the symbolic character string of the constant.

SUBROUTINE SNFLPT

Mnemonic Origin: Scan Floating Point constant

Classification: FFE scan service routine

Purpose: Extract Floating Point constant constructions from elementary entries in Parsing Tables.

Operation: Upon entry, the Parsing Tables are positioned to the first element of a construction which has been identified as a Floating Point constant. The constant may be either a single or double precision constant.

The mantissa entry is extracted from the Parsing Table entries and placed in the lexical item. These entries consist of the integer, decimal point, and fraction components of the mantissa. Not all components are necessarily present in the mantissa description.

The optional explicit exponent specification is then processed. Since the exponent precision character may be run on with the exponent value, the first character of the exponent specification is extracted to determine the Intermediate Symbol String code to be assigned to the resulting Floating Point constant construction.

If an explicit exponent is present, the characters of the exponent specification are extracted from Parsing Table entries and accumulated with mantissa characters in the lexical item.

After extracting the characters of the Floating Point constant, the current Parsing Table entries are replaced with a single entry. This requires deleting the elementary entries in the Parsing Tables, reserving space to insert the collected characters and new ISS code.

SUBROUTINE SNHOLL(INCHR)

Mnemonic Origin: Scan Hollerith Literal strings

Classification: FFE Scan process service routine

Purpose: Process Hollerith literal strings to construct Parsing Table entries for character data.

Operation: Upon entry, the start of a Hollerith literal string sequence has been detected from incoming card data. The literal string may be of the character count/string form or quote mark delimited form. The passed parameter is used to distinguish the two forms.

If the form is the count/string form, the count is presented as entries in the lexical item. The state of the lexical item buffer is examined to determine if the string can be accommodated in the available space. If not, the character string will be truncated to the symbols which can be stored. Sequential characters are obtained from the card image and inserted in the lexical item until the character string is spanned or the available space is exhausted. If an end of statement code is encountered, card image data is exhausted before the specified number of characters were processed. If more characters are present than available space, additional characters are discarded to position the scan to the character following the character literal string. Control is then transferred to the storage process to make Parsing Table entries.

If the character string is a quote delimited form, characters are extracted until the terminal quote mark is encountered. The delimiting mark may be either a single quote mark (i.e. apostrophe) or a double quote mark (i.e. formal quote mark). The type of delimiting mark is recorded. To accommodate the use of a double delimit mark as a single character, the process looks ahead one character. If two occurances of the delimit are detected, only one character is stored and the second mark is discarded.

If the lexical item space is exhausted, the process records a truncation condition. The overflow situation simply discards characters without placing them in the lexical item; major processing activities continue to scan past the character string of the literal to achieve proper positioning for the next card item.

If an end of statement code is found while processing delimited forms, the terminal mark is missing on the card image. This event causes termination of the processing loop after issuing a message that the symbolic form is erroneous.

After detecting the terminal mark in the normal string form, the Scan Buffer is backed up one position to correct for character look ahead. Control is transferred to the storing procedure for making Parsing Table entries.

After detecting the terminal mark in the normal string form, the Scan Buffer is backed up one position to correct for character look ahead. Control is transferred to the storing procedure for making Parsing Table entries.

The storing procedure reports truncation effects detected during string processing and makes Parsing Table entries. The ISS code is stored in the next position and the character string held in the lexical item transferred to the next Temporary Symbol Table position.

Parameters: INCHR - input character which was detected by the calling routine indicating a Hollerith constant present.

Special Remarks: Note that the Hollerith scan process differs from other scan routines in selecting the next sequential character rather than the next nonblank character.

SUBROUTINE SNNUMB(INCHR)

Mnemonic Origin: Scan Numbers

Classification: FFE Scan Service routine

Purpose: Scan lexical items beginning with a number.

Operation: Upon entry, a lexical item is discovered which begins with a numeric entry. The item may be a simple integer (statement label or constant) or a component of a more complex form. If the integer is a simple constant, the character string is processed and stored by this routine. If the integer is a portion of a complex form which is processed in the preliminary scan procedure, control is passed to another routine for completion of the processing.

Characters are retrieved from the card image data and placed in the lexical item until a nondigit character is detected. The terminating character is examined to determine if control should be transferred to another routine or the item stored as an integer entry.

If the terminating character indicates the integer is a count specification of a Hollerith character string, control is passed to the Hollerith string process.

If the terminal character indicates the item is a nondecimal constant representation, control is passed to the nondecimal constant process.

If a simple integer item is to be stored in the Parsing Tables, the entry is made inserting an ISS code and Temporary Symbol Table entry. The Scan Buffer is backed up to recover from overrun created by extracting the terminal character.

Parameters: INCHR - input parameter contain the character used to determine that an integer string is in progress.

Special Notes: SNNUMB is used to process statement labels. In FACES, a statement label is simply an integer as the first entry of the statement; no valid FORTRAN statement begins with an integer constant.

Labels receive no special treatment in processing. They are, however, not permitted to imply a count for Hollerith or nondecimal constants. If a statement label is present, it will be the first Parsing Table entry.

SUBROUTINE SNPERD(INCHR)

Mnemonic Origin: Scan Period delimited forms

Classification: FFE scan service routine

Purpose: Scan lexical item forms which are delimited by a pair of periods.

Operation: Upon entry, a form may be present consisting of a variable character string enclosed in a pair of periods. This form may be a logical constant, relational operator, or logical operator.

The form is first identified to determine if the structure .V. is present in the Parsing Tables. The last period is passed by parameter. The structure .V is found in the last two ISS entries. If the form is not found, the period is stored in the Parsing Tables and control returned to the calling routine.

If the form .V. is detected, the character string associated with the V entry of ISS is examined from the Temporary Symbol Table entry. The symbolic content of this entry is compared to a series of read-only templates established in Block Data. If a template matches, the entry is identified as a period delimited form.

Associated with the matching template is an ISS entry code for the form. These values are also assigned in Block Data.

If the ISS entry requires a Temporary Symbol Table entry to be stored, the character string of the matching character string

is surrounded by periods in the lexical item and replaces the current entry in TSTAB. If no entry is required, the current TSTAB entry is remove.

ISS entries are replaced by the indicated code whether a Temporary Symbol entry is required or not. This is accomplished by removing the current elementary entries from the last ISS table positions prior to storing the new ISS code.

Parameters: INCHR - contains the symbol period detected by the calling routine.

SUBROUTINE SNZPRO(INCHR)

Mnemonic Origin: Scan Z form lexical items.

Classification: FFE Scan Service routine

Purpose: Process nondecimal constant forms.

Operation: Upon entry, a lexical item is detected which may be a nondecimal constant form. (These forms were denoted as Z forms historically in the system development.) The lexical item may be identified definitely as a nondecimal form or may be a variable specification which appears to be similar to a nondecimal constant form. If a nondecimal constant can be identified conclusively, the form is accepted as a nondecimal form. If the form is ambiguous with a potential variable, control is passed to the alphabetic processor with processed characters residing in the lexical item.

Three basic forms of nondecimal constants are processed:

1. Count/indicator character/nondecimal string
2. Indicator character/nondecimal string
3. Nondecimal string/indicator character

Notice that type 2 is the same form as a variable declaration. It can only be distinguished from a variable name by having more characters than a variable name in the target FORTRAN or by being used in a constant context (e.g. a DATA statement constant list). Since FACES uses a blind scan, the context of the statement is not known; the symbol is recognized as a variable character string

by the scan process with correction required by later processing if this is a constant.

Different forms are distinguished by the calling routine and contents of the lexical item when control is initiated. If the form is a count followed by an indicator character Z, and unambiguous nondecimal constant form of hexidecimal characters is found. The count, present in the current lexical item entries, is converted to an integer value. Subsequent characters are extracted from the card image data until a symbol other than a digit or alphabetic character is detected or the count is exhausted. Notice that the base of the symbols is assumed correct; valid base 16 alphabetic characters are not checked.

Character symbols of the constant are inserted in the lexical item if sufficient space remains. If the lexical item space is exhausted, the truncation indicator is set and the character is discarded. This technique allows the scan to proceed past the truncated characters and position to the proper character of the next lexical item.

The lexical item is then inserted in the Parsing Table entries. An ISS code is selected for the item based upon the memory requirements of the target machine. The code selected indicates the memory required for a constant of the indicated length in words of target machine storage.

"Leading character" indicated lexical items may begin with either the character O or Z; O indicates an octal constant; Z indicates a hexidecimal character string representation. The only difference in processing these two forms is the conditional test for subsequent characters of the constant. Characters are accepted from the card image until an unacceptable symbol for the constant string is detected. Received characters are placed in the lexical item.

After terminating character acceptance, the length of the accepted string is examined. If the number of characters is less than the number of characters permitted in variable names of the target FORTRAN, the character string must be classified as a variable name for processing. In this event, control is passed the alphabetic service routine with the extracted characters residing in the lexical item and the terminating character passed by parameter.

If the accepted string is longer than a variable name in the target FORTRAN, the string is stored as a nondecimal constant. An ISS code of integer is attached to the symbol string.

The third form is a digit string terminated by an indicator character. When control is gained, the lexical item already contains the digit string. The indicator character is appended to the lexical item and the item is stored in a Parsing Table entry assigning an integer ISS code.

Parameters: INCHR - input parameter containing the character detected by the calling routine indicating the potential presence of a nondecimal constant.

FUNCTION SRCHDI(NAME1, NAME2)

Mnemonic Origin: Search Directory for Name

Classification: AIR General Purpose Utility

Purpose: Search for name in Directory.

Operation: Search Directory character string contained in NAME1 and NAME2. If character string found, set SRCHDI to location of string. If not found, set SRCHDI to zero.

Parameters: NAME1 - Input

NAME2 - Input

SRCHDI - Output

SUBROUTINE STATL

Mnemonic Origin: Statistics on local table usage

Classification: FFE maintenance support

Purpose: Provide statistics on usage of local tables to configure table allocation for typical runs.

Operation: Upon entry, the Local Tables of a module have been produced. Information is required on the space used in the current Local Table allocation to determine if insufficient space is allocated to a Local Table or significant unused space has been allocated.

The current used of Local Tables for the individual module is required to be reported and a cumulative indicator of the largest space used by any routine needed. Since the Symbol Table is a hash coded structure, the contents of the table must be reviewed to determine how many entries are actually active.

The current table usage values are then combined with previous results from other modules processed during this run to obtain the peak value for each table on the run.

Special Notes: Note that the cumulative value variables are local to this routine. They are set by a resident DATA statement. Since this module is only a maintenance support routine, capacity to participate in overlay is not considered significant. At most, a potential for incorrent accumulated value is present if the module is overlayed.

SUBROUTINE STOW(PLACE)

Mnemonic Origin: Store item in Temporary Symbol Table

Classification: FFE Scan support routine

Purpose: Place the contents of the lexical item in a Temporary Symbol Table entry.

Operation: Upon entry, the lexical item is occupied by characters which constitute an entry in the Temporary Symbol Table. These characters are to be packed into A4 format and inserted in a Temporary Symbol Table position. The position is either the next available position at the end of the table or a specific position indicated by the calling routine.

The specified table position is first examined for validity. If an explicit table location is specified, the specification is examined for valid range. If the table position is defaulted to the next available position at the end of the table, space is allocated for the entry.

If table space is available to accommodate the entry, insertion of the item is performed; if the table does not have space to accommodate the entry, an error message is issued and the last table position is overwritten with the lexical item.

Insertion of the lexical item is dependent upon the length of the character string. If the item is too long to fit in the main table entry, space is required in the overflow table. If storing the lexical item data would result in exceeding the overflow table space, the lexical item is truncated to fit available

space. To prevent exceptionally long items from consuming all the overflow entries, the lexical item is permitted to occupy only half the remaining space of the overflow table. Notice that truncating the item will eventually result in items being reduced to lengths of 8 characters or less. This permits shorter items to use main table entries as truncated forms when the overflow space becomes shorter. Truncating the item is accomplished by simply reducing the pointer to the last nonempty entry of the lexical items.

After allocating table space for the lexical item, the item is inserted in the Temporary Symbol Table. If the lexical item will fit in the main symbol table entry, the character string is converted to a 2A4 format entry and inserted in the table position. If the item is longer than 8 characters, the main entry is set to an empty first word and pointer to the overflow table space allocated for the character string. In the overflow table, the first entry is a count of words containing characters belonging to the entry. This is immediately followed by the characters packed in A4 format.

Parameters: PLACE - input parameter indicating the table location to be used in storing the item. Contains a positive integer value if a specific table address is required by the calling routine. If the next available table location is desire, the value zero is passed.

SUBROUTINE STMEND

Mnemonic Origin: Statement End

Classification: FFE parsing support routine

Purpose: Terminate the processing of a FORTRAN statement.

Operation: Upon entry, a FORTRAN statement has been completely processed. A call to this routine is an assertion that all entries of the Parsing Tables have been consumed. If an alien statement construction has resulted in premature termination of the statement, some additional text may not be processed.

If the Parsing Tables are positioned to the end of statement code, the statement was completely processed. Otherwise, only a portion of the statement was processed and a truncated statement message is required. If the statement is truncated, the Parsing Tables are positioned to the end of the structure to flush unused text.

SUBROUTINE SUCGEN

Mnemonic Origin: Successor Generation

Classification: FFE table generation routine

Purpose: Generate list of successors for statements recorded in Node Table.

Operation: Upon entry, program statements (nodes) have been entered in the Node Table. Program transitions have been converted to statement numbers in the transition pairs table. It is now required to enumerate the successor nodes to each statement.

Successors to a statement are both the explicit successors recorded in the transition pairs table and the implied successors resulting from normal statement to statement transition in non-branching flow.

The transition pairs table entries are first sorted on predecessor entries. This causes the successors of a statement to appear in adjacent table positions and forces special codes to the bottom of the table. Nodes are then examined one at a time from the first node to the last to develop a list of successors for each node.

Node table entries are processed from the first entry to the next to last entry. Since the last node of the program has no implied successor (i.e. no next statement is present), this case is treated separately. First the implied successor is inserted in the successor list. The next statement is an implied

successor if the current node is not a transfer of control statement or a program termination statement.

After processing implied successor entries, the explicit successors, if any, are added to the successor list. These entries may be either transfer targets of branching statements or special code transitions for calls, statement function references, or program boundary transition.

To construct successor lists, entries are made using the current pointer to the Successor Table. The last nonempty pointer value is preserved to permit computing the number of successors entered and protect against overflow of the Successor Table space. After all successors for a node have been entered, the count is computed for the number of entries and the pointer/counter entry made in the Node Table position. If no successors are present for the node, the pointer and counter are set empty in the Node Table entry. After making entries for the node, the last nonempty pointer to the Successor Table is advanced to the end of the Successor Table list.

The last node table entry is processed by considering only the explicit transfers from the node. These entries are made in a similar fashion to the Successors Table entries produced for other nodes.

If Successor Table space is exhausted while processing the list, an error process is executed. The error process distinguishes between space exhaustion while processing the successors of a statement and space exhaustion which occured on the first successor of a statement. To issue the error message, the card count must be moved to the node being processed. This is required since the construction of successors occurs after the program has been processed; the current values of the card count are positioned to the end of the module.

Finally, the error process clears remaining Node Table successor entries to an empty state.

SUBROUTINE Txxx(TAB, BF)

Mnemonic Origin: Transition from Table xxx to Table TAB

Classification: AIR General Purpose Utillty

Purpose: Extend pattern search from list in one table to list in another table.

Operations: Using information concerning list in table xxx, find entry point in list in table TAB. Place location information in Table Name Register (TR), List Indicator Registor (LR), and Pointer Register (PR). If entry point does not exist, set registers to zero.

If entry point found, set Branching Flag BF to one; else, BF to zero.

Parameters: TAB - Input

BF - Output

xxx		TAB
COM	→	LIN
DIR	→	IS, SH, SYM
IS	→	ISD
ISD	→	DIR
LIN	→	DIR
NOD	→	PRE, SUC, USE2
PRE	→	NOD
SH	→	SHD

xxx		TAB
SHD	→	DIR
SUC	→	NOD
SYM	→	DIR, USE1
USE1	→	NOD, SYM, USE2
USE2	→	NOD, USE1

See also TT, /SPEREG/, "Traversing Lists", and "Legal Table to Table Transition".

SUBROUTINE TRACHI(MODBGN, FOLBAK, COND)

Mnemonic Origin: Trace System Hierarchal Paths

Classification: AIR Special Purpose Utility
(Referenced only during search for cyclic calling sequence).

Purpose: Follow calling sequence paths.

Operation: Trace calling sequences, one module at a time, starting at module specified at location MODBGN in Directory. FOLBAK indicates whether path is being followed or backtracked. COND indicates condition under which control was returned to calling routine.

Algorithm: See Source Code Listing.

Condition Codes: See Source Code Listing.

Parameters: MODBGN - Input
FOLBAK - Output
COND - Input/Output

See also "Calling Sequence Path Tracing" and Trace Stack.

Possible table to table transition:

COM	→	LIN
DIR	→	IS, SH, SYM
IS	→	ISD
ISD	→	DIR
LIN	→	DIR
NOD	→	PRE, SUC, USE2
PRE	→	NOD
SH	→	SHD
SHD	→	DIR
SUC	→	NOD
SYM	→	DIR, USE1
USE1	→	NOD, SYM, USE2
USE2	→	NOD, USE1

See also IE, "AIR Basic Search Technique", "Pattern Searches", "Traversing Lists", and "Legal Table to Table Transitions".

SUBROUTINE TRANS(MOD)

Mnemonic Origin: Transition in program

Classification: FFE table construction routine

Purpose: Record program transitions while processing FORTRAN statements.

Operation: Upon entry, a transition is discovered for the current statement being processed. The transition may reference a symbolic location in the program; in this event, the Symbol Table is currently positioned to the referenced symbol.

The transition may be an explicit branch caused by the current statement (e.g. branch target of an IF statenent), a program reference to a procedure (e.g. subroutine or statement function reference), program boundary condition (e.g. entry point or termination), or a branch condition imposed on a future statement (e.g. DO statement). The type of transition is indicated by the control parameter passed.

Normally, one entry is made in the transition pairs table for each call to TRANS. In some instances, two entries are required to record the transition. The exact linkage is processed by a series of independent cases which establish the predecessor, successor, and postprocessing specification to be recorded in the transition pairs table entry. A predecessor of the current statement, requiring one TRIP entry, without postprocessing is

is established by default. These values are the most commonly required but may be overridden by individual cases.

After establishing the required entry values, space is examined in the transition pairs table. If sufficient space remains, the entries are made at the end of the table. At two bit postprocessing code is attached to required entries to record the need to convert references to statement labels to node numbers. Since a statement label may be referenced before it is defined, no attempts are made to convert the definition point until the end of the program.

Parameters: MOD - control paramter used to indicate the type of transition to be recorded.

Unusual Cases: DO statements: Since the ANSI standard indicates DO statement bodies are always executed once, the graph transitions generated require two entries. When the DO statement is recognized and the statement label recorded, transitions are made from the terminal statement back to the DO statement. The normal statement to statement implied transition is assumed to achieve loop fall through from the terminal statement. Note that the transitions are recorded before the terminal statement is processed. Also note that the DO terminal statement may terminate multiple loops.

Logical IF statement: Logical IF statements with a conditional statement execution are processed

as two statements. While processing the IF condition portion, the transition is recorded from the IF statement to the next statement and from the IF statement to the statement after the next statement. Statement numbers to be assigned are implied from the current statement number.

SUBROUTINE TRPSRT(COL)

Mnemonic Origin: TRIP Sort

Classification: FFE table manipulation routine

Purpose: Sort the contents of the transition pairs table (TRIP) to order the contents for processing.

Operation: Upon entry, the Transition Pairs Table (TRIP) contains entries indicating program transitions within the module. Entries have been converted to statement numbers and special codes. The entries are to be sorted in an ascending fashion to process the transitions of individual statements.

This routine is capable of sorting on either the predecessor or successor entries. Since table entries have been converted, the flag field of entries are now cleared to zero valued bits. As a result, all entries contain positive values at this point in the process.

The passed parameter controls whether the contents are ordered by successor entry contents or predecessor entry contents. The control parameter is used to establish the selected column (i.e. either predecessor or successor) of the table structure. The other column becomes the nonsort column.

Sorting is accomplished by a standard shuffle sort. Entries are examined one at a time looking for out of order entries between the current entry and the next entry. If an out of order entry is detected, the entry is moved back in the list to a position where proper order is achieved.

Note that the sort process maintains the relative order of entries. That is, if two entries contain the same value as the sort key, the relative order of the entries is maintained.

Note also that special codes are numerically much larger values than statement numbers. Therefore, the sorting effectively moves special codes to the bottom of the list.

Parameters: COL - input control parameter which determines whether the sorting is accomplished on predecessor or successor entries of the table contents.

SUBROUTINE TT(TAB, BF)

Mnemonic Origin: Table to Table Transition

Classification: AIR General Purpose Utility

Purpose: Allows access to entire lists in tables, from first list element to last.

Operation: Traverse list in table TAB, one list element at a time. Which list in table TAB is determined by information at top of Control Stack. (Transition is from table indicated at top of Control Stack to table TAB.) If list entirely traversed, branching flag BF is set to two; else, BF is set to one.

Algorithm: TT performs one of two complex operation, depending on whether Forward-Backward Register FBR indicates forward or backward..

Forward - Find initial entry point into list in table TAB from table indicated by top of Control Stack. Place this information and associated information into top of Control Stack.

Backward - Find next list element. Update information at top of Control Stack.

Parameters: TAB - Input

BF - Output

SUBROUTINE TYPCHK(NAM, ITYP)

Mnemonic Origin: Type check

Classification: FFE table generation routine.

Purpose: Type symbolic names extracted from FORTRAN text using the first letter of the name.

Operation: Upon entry, a symbolic name has been encountered for which a type code is required based upon the first character of the name. The first character is extracted from the symbolic name and converted to a numeric index from 1 to 26 corresponding to the alphabetic characters A to Z. Using this code, the typing vector is addressed to access the appropriate type code.

Protection code is provided to protect against invalid character strings and possible contamination of the typing vector. If the presented character string begins with a symbol other than an alphabetic character, integer type is assigned. If the typing vector addressed contains an invalid type code, default type is assigned using FORTRAN default types based on the first character.

Parameters: NAM - first characters of the symbolic name in A format.

ITYP - output parameter receiving the assigned type code based upon the character string presented.

SUBROUTINE UNINT (ARRAY, ARRSIZ)

Mneumonic Origin: Uninitialized Variable

Classification: AIR Query

Purpose: Searches for uninitialized local variables.

Operations: All paths between entry point of module and variable under investigation are examined.

Program boundaries are not crossed. It is assumed that any variable appearing in parameter list is assigned value on other side of program boundary.

Array ARRAY has ARRSIZ entries which indicate whether warning messages may be produced for primary or secondary listings.

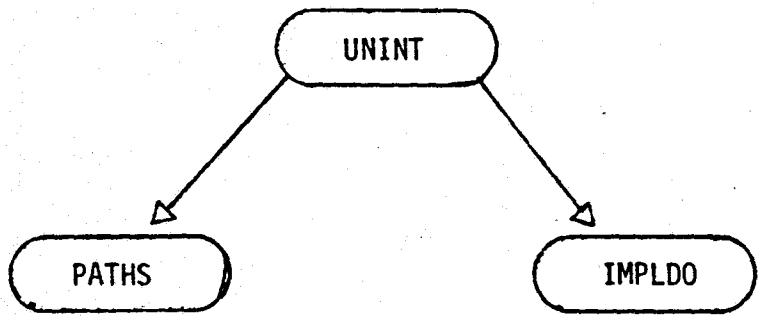
Algorithm: See Source Code Listing.

Codes: See Source Code Listing.

Parameters: ARRAY - Input

ARRSIZ - Input

See also /PAT/ and PATHS.



Uninitialized Local Variable Search,

excluding General Purpose Utilities

SUBROUTINE UPNOD

Mnemonic Origin: Update Node Table

Classification: FFE table generating routine

Purpose: Place the current statement description in the next Node Table position.

Operation: Upon entry, the current statement contains a partial description of a node in the program. Successor and predecessor information is still lacking in the description.

A Node Table entry is made using the partial description to establish the statement type, first Use Table position related to the node, and module relative card counts for the statement. Since the statement number is advanced sequentially, the Node Table insertion position is established from this counter. To protect against possible errors, the pointer to the last entry is established using the cumulative maximum of statement numbers found to this point in the process. With this mechanism, gaps in the sequence will at most produce empty statement entries in the Node Table.

SUBROUTINE UPSYM(NAME, LENGTH, TYPE, CLASS)

Mnemonic Origin: Update Symbol Table

Classification: FFE table generation routine..

Purpose: Insert a symbol in the Symbol Table

Operation: Upon entry, a symbol is presented for insertion in the Symbol Table. The type and class are also provided for insertion to describe the symbol. The pointer to the Symbol Table has been positioned to the proper insertion point.

Insertion processing depends upon the length of the symbol entry. If the symbol is longer than the main table entry can accommodate, the character string is to be stored in the Symbol Overflow Table. If insufficient space is available in the Overflow Table, the symbol is truncated to a size which can be accommodated in the main table entry. Since the truncated representation may match a symbol previously recorded, (i.e. previously truncated) the Symbol Table contents are then investigated for a matching entry; if one is found, the insertion process is suspended.

Oversized entries are inserted with a pointer to the overflow entry data structure in the main Symbol Table entry. The type and class code of the symbol are placed in the main Symbol Table structure; only the character string is diverted to the overflow structure.

Overflow Table entries are made with a count heading the character string of the symbolic characters. The count indicates how many words follow containing the oversized character string.

Symbols which will fit in the main Symbol Table entry are directly inserted in the entry.

Parameters: NAME - symbolic characters of symbol packed in A4 format.

LENGTH - length of the symbol expressed as the number of integer words needed to accommodate the character string.

TYPE - type code to assign to Symbol Table entry.

CLASS - class code to assign to Symbol Table entry.

SUBROUTINE USERQ

Mnemonic Origin: User Queries

Classification: Control Driver command card interpretation

Purpose: Construct the queries to be executed by the AIR Subsystem.

Operation: Upon entry, a QUERY command has been recognized. The command card has been processed through the entry QUERY. User specifications are to be examined to establish the queries to be executed. Executed queries are communicated to AIR routines through values set in the List Table.

The next command item is requested from the command card and List Table is set empty. Note that the List Table is an AIR data structure. Therefore, the length of the List Table is controlled in INTAIR. To accommodate the ambiguous and possible flexible length, a local variable is used to govern insertion in the List Table entries. Proper operation is insured so long as the local variable is not longer than the List Table data space.

Query lists are processed by cases established through examination of the first option field on the QUERY card. The cases are implemented as a ladder search of potential entries. Processing options requires the insertion of one or more categories of queries in the List Table. The queries available are retrieved from read only data vectors set by BLOCK DATA. Categories of available queries are LOCAL, GLOBAL, and SYSTEM.

System queries are internally required queries needed to cause Global Tables to be constructed.

If the QUERY card has no option specified, all queries are inserted in the list. If local tests are requested, the default grouping of local tests is inserted in the list. Notice that local queries are not all inserted by the LOCAL test; expensive path tracing queries are requested only by explicit request. The boundary is established by a pointer to the Local query list data structure.

If Global queries are requested, both system and global queries are inserted in the query list. This will cause global tables to be constructed in case they are needed by global queries. The system queries must appear prior to global tests in the list for proper operation.

If an ONLY option is specified, the user is providing queries to be inserted in the list. System queries are inserted in case a global query is presented. In processing user specified lists, queries are extracted one at a time from the list until no more commas are found after the last entry. Each specified query is compared to the list of available numbers to verify the validity of the specified number. Queries are inserted in the List Table in the order received from the command card.

After establishing the query list, the EXCEPT option is examined. If an exception list is present, queries in the List Table are examined and turned off. Turning off a query is accomp-

complished by setting the corresponding entry empty.

After establishing values in the List Table, the first Map entry is set to describe the list to the AIR system for interpretation of instruction.

Special Notes: System queries will only result in table construction actions if the tables are not already present. Thus, the cost of executing system queries is negligible if the tables are already present. For this reason, system queries are produced whenever the potential need is present.

SUBROUTINE VFYGHD(ABORT)

Mnemonic Origin: Verify Global Header

Classification: Control driver initialization procedure

Purpose: Verify the contents of the global header on the table file as containing valid entries compatible with FACES operation.

Operation: Upon entry, the global header of the Table File has been acquired for a run. To verify that a proper file has been attached as the Table File and that data contents are compatible with the current version of FACES, the contents of the global header are compared to data values set by the current configuration.

Some header errors may result in erroneous or questionable results; others are likely to produce serious problems. Less severe problems are simply reported to the User; dangerous situations cause the system to cease operation.

The version and modification level of the current system are compared to the system which produced the analysis table. Similarly, the Host machine used for table production and Target machine for which the system was adapted are compared to those for which the tables were produced. Differences are noted but the run is allowed to continue.

Global header entries for Source Code Catalogue and Analysis Table File positions end locations are examined. If

negative values are found, the run is aborted since an obviously invalid file is being presented as the Analysis Table File.

Similarly, invalid entries for the lengths of Local or Global Tables and pointers to the last nonempty entries of Global Tables will result in abortion of the run. These indicate an incorrect file has been presented as the Analysis Table file.

Parameters: ABORT - output parameter set TRUE to indicate the the run should halt due to configuration incompatibilities. Set FALSE if the global header contents are compatible with the current configuration.

SUBROUTINE WRFMSG(ONCE, MNUM MCHR1, MCHR2)

Mnemonic Origin: Write FORTRAN Message

Classification: FFE error reporting routine

Purpose: Write a message to the Flag File to inform user of error condition which limits analysis.

Operation: Upon entry, the FFE has discovered a condition which is to be reported to the user. To communicate this information records are placed on the Flag File for use in generating reports. The message is associated with the current statement being processed.

The message to be transmitted may require a single flag or several flags. A single entry is recorded if the total message can be transmitted with one record of information. The number of flags is governed by the amount of data to the transmitted in the message.

Data for transmission has been placed in the message COMMON Block prior to calling this routine. The data may be empty (i.e. have length zero) if flag indentification information is sufficient to explain the problem.

Because of implementation technique, some errors may occur many times while processing a module; these messages are significant only on the first report. For example, if a table

overflows, the error may be reported with each attempt to store an item. To suppress this unnecessary repetition, a vector is established to record the first occurance. The calling routine identifies messages which require suppression. If the parameter is zero, the message is to be reported on each occurance; if the parameter is a positive integer, the message is to be reported only if it has not occured previously.

Processing begins by examining the control parameter to determine if the call should actually produce error messages. If the message should be reported, card information for the error is inserted in the message variables. Card information is obtained from the current statement description.

The flag number (integer value) and descriptive characters (alphabetic characters) are inserted in the output message. If the length of the message is zero, neutral values are inserted as data fields in the flag to be written.

Using the constructed information, as many flags are written as data items contained in the constructed message. If more than one flag is written, the internal order is advanced with each flag output.

After processing, the message length is reset empty to condition the message COMMON Block for the next message.

Parameters: ONCE - control parameter used to suppress unnecessary error reports on errors which occur more than once in a module. Contains either the index to the suppression array used to record error occurances or the value zero indicating no suppression is to be applied.

MNUM - integer value for the flags to be written.

MCHR1, MCHR2 - Alphanumeric characters to be written as the alphanumeric description of the flag.

SUBROUTINE XFRISS

Mnemonic Origin: Transfer Intermediate Symbol String

Classification: FFE error reporting routine

Purpose: Insert the contents of the Intermediate Symbol String in the error message COMMON block to identify the error location of a processing error.

Operation: Upon entry, the FFE has discovered an error in the FORTRAN code which limit processing. An error message is to be issued to inform the user of the error location. The Parsing Tables are currently at the error position. To communicate the error location, the contents of the ISS entries in the vacinity of the error are to be inserted as error message data.

The data transferred is limited to the six entries of ISS including the error position to prevent excessively long message information. These entries are extracted and placed in the alphabetic data fields of the error message COMMON structure. The associated integer data field is set to zero values. Finally, the length of the inserted message is set to communicate how much data is to be transferred.

SUBROUTINE XFRLEX

Mnemonic Origin: Transfer Lexical Item

Classification: FFE error reporting routine

Purpose: Present the contents of the Lexical Item for user information to explain an error message.

Operation: Upon entry, the lexical item contains character data which supports an error condition detected. The information in the lexical item is transferred to the error reporting COMMON Block to transmit this information to the user on the report to be generated.

The character string space requirements are examined first to determine if the active lexical data can be accommodated in the error message reporting data structure. This analysis predicts the space needed for packed items of characters. If sufficient space is available in the recording vector.

Elements of the lexical item are then extracted and packed in A4 format for error reporting. These items are placed in the alphabetic data fields of the error message, setting the corresponding integer data fields to zero values. If no data is transferred in this process or a nonmodulo 8 number of characters are processed, remaining alphabetic fields are set to blank characters.

SUBROUTINE XFRSYM(TSTLOC, TYPE, CLASS)

Mnemonic Origin: Transfer Symbol

Classification: FFE table generating routine

Purpose: Transfer a symbol from the Temporary Symbol Table to the Symbol Table.

Operation: Upon entry, a Temporary Symbol Table entry is to be transferred to the Symbol Table. The type and class assigned to the entry is presented by parameter.

The temporary symbol entry length is examined to accomplish the transfer. If the symbolic characters are located in the main Temporary Symbol Table entry, the characters are extracted and passed to the Symbol Table recording routine.

If the characters are located in the Temporary Symbol Overflow table, the character string is accessed and passed from the overflow entry.

Parameters: TSTLOC - Temporary Symbol Table location containing the symbol to be transferred.

TYPE - type specification assigned to the symbol being recorded.

CLASS - class code assigned to the symbol being recorded.